Organization of structured information for mechanized retrieval operations and some related efficiency considerations.

Twite, Martin J.

ORGANIZATION OF STRUCTURED INFORMATION FOR MECHANIZED RETRIEVAL OPERATIONS AND SOME RELATED EFFICIENCY CONSIDERATIONS

MARTIN J. TWITE

ORGANIZATION OF STRUCTURED INFORMATION FOR MECHANIZED RETRIEVAL OPERATIONS AND SOME RELATED EFFICIENCY CONSIDERATIONS

* * * * *

Martin J. Twite, Jr.

ORGANIZATION OF STRUCTURED INFORMATION FOR MECHANIZED RETRIEVAL OPERATIONS AND SOME RELATED EFFICIENCY CONSIDERATIONS

by

Martin J. Twite, Jr.

Lieutenant Commander, United States Navy

Submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE
IN
ENGINEERING ELECTRONICS

United States Naval Postgraduate School
Monterey, California

1 9 6 4

PS Archive
964
mith, M

ORGANIZATION OF STRUCTURED INFORMATION FOR MECHANIZED RETRIEVAL OPERATIONS AND SOME RELATED EFFICIENCY CONSIDERATIONS

by

Martin J. Twite, Jr.

This work is accepted as fulfilling

the thesis requirements for the degree of

MASTER OF SCIENCE

IN

ENGINEERING ELECTRONICS

from the

United States Naval Postgraduate School

ABSTRACT

Information may be stored in the memory or the memory extensions of digital computers. The ease with which this information may be manipulated and retrieved is greatly complicated if it is of the non-numeric type. However, techniques are known which facilitate the solutions to these information storage and retrieval problems. These techniques allow the optimization of efficiency according to one or more criteria. Various list languages are introduced and the overall information storage and retrieval problem is outlined. Four basic storage allocation systems are discussed and their relative merits compared. Tree structures embodying the best features of the four basic systems are described and evaluated. Efficiency computations from the standpoint of search time and the search-time, storage-space product are presented. A specific tree structure called a trie is introduced as a compromise between the two extremes of the balanced tree and the fully elided unbalanced tree. Finally, a technique called Fibonaccian searching is presented as a method to conserve both storage space and time.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Program Symbols

$\underline{a}$	accumulator
$\underline{c}$	coarseness vector for dictionary search
$\underline{e}$	vector of all 1's
$\underline{i}$	identity permutation vector
$\underline{M}^i$	ith word of memory
$\underline{p}$	permutation vector
$\underline{s},\underline{x}$	argument item
$\underline{t}$	temporary storage register
$\underline{u}$	mask vector
$\underline{v}$	logical vector to select item or character portion of word
$\underline{w},\underline{y},\underline{z}$	logical vectors to select chain address portion of word $\underline{w}$ heir chain, $\underline{y}$ sib chain, $\underline{z}$ character chain
*	error exit
λ	end-of-chain marker
$k \leftarrow \lfloor x \rfloor$	$k \leq x < k + 1$ floor
$k \leftarrow \lceil x \rceil$	$k \geq x > k - 1$ ceiling
$\underline{c} \leftarrow \underline{u}/\underline{b}$	$\underline{c}$ is obtained from $\underline{b}$ by suppressing each b_i for which $u_i = 0$
$\bar{\underline{u}}$	the complement of vector $\underline{u}$
$M^{\text{Row}}_{\text{column}}$	Matrix super and sub scripts

List of Symbols and Parameters

TABLE I

Parameters

a	number of bits to specify an address
b	number of bits to specify a character
c	number of character positions which are chained
d	average number of characters per set (average degree of tree)
e	average number of characters in the function part of an item
f	number of item times to compute function f
g	number of items governed by a particular node
h	number of character sets (height of tree)
i,j,k	various intermediate and running parameters
m	number of members of each character set
n	number of items in the file
r	storage ratio
s	number of items satisfying a multiple-match search
t	search time
w	number of bits per memory word
z	number of zeros in mask vector

List of Symbols and Parameters

TABLE I

(continued on next page)

1. Introduction.

It is the purpose of this paper to bring together in a single document a description of some of the schemes which have been proposed in the field of Information Storage and Retrieval. Further, it is intended that the emphasis be on the efficiencies of the various schemes and how these efficiencies are modified by interaction during combination.

Automated information storage and retrieval has for some time been a dream of scientists and engineers. A most provocative article was written by Vannevar Bush just after World War II describing the "library problem" and a proposed solution. [1] Most of what was proposed could then only be classified as speculation but we have seen that this speculation was well based. His proposed private file and library, which he named the "memex", is within the capabilities of our technology today. The bar to its realization is the complexity of the indexing and manipulation of non-numerical information. Much work has been done on the problem, but we are still a long way from a completely automated system. It is not suggested that what follows here is an exhaustive compilation of the state-of-the-art. It is rather a sampling to indicate the current status of some of our attempts at completely mechanized information storage and retrieval.

It is intended that Section 2 will provide an introduction to a widely used device for manipulation of non-numeric information, i.e. the list. Also it briefly describes several computer languages which use lists. Section 3 describes the information storage and retrieval problems of 1) automatic determination of document

content, 2) organization of structured information within a computer, and 3) evaluation of systems. Specific file processing operations are discussed in Section 4 in order to provide a common basis for evaluation of the storage allocations described in Sections 5 and 6. Sections 5 and 6 contain programs which describe the searches and are intended to convey a feeling for the type and amount of processing required. Section 7 shows how it is possible to design trees to minimize the time necessary to search them while at the same time holding the tree's storage requirements to a minimum. A comparison is made in Section 9 between the storage requirements of trees representing both extreme and mean types. Finally, an efficient method of searching an ordered list in a machine not capable of a binary shift is given in Section 9.

2. Hueristic programming and the development of list languages.

The application of digital computers has been most successful thus far in areas where the manipulation of numeric quantities is the primary operation. More recently, attempts are being made to handle more complex and ill structured problems such as theorem proving, playing chess and checkers, or performing various symbolic calculations like differentiation and integration. The motivations behind this work range from a desire to extend the capabilities of computers to the desire to understand how humans think, learn, and solve problems. [24]

In a non-numeric problem the contents of words in memory are not treated as though they contained numbers (although in fact they do). Rather, the contents are treated as alphanumeric strings of characters or symbols. For example, they might be algebraic expressions, coded information, or English words. For this reason the term symbol manipulation is sometimes used to characterize the procedures used. [35]

A fairly common characteristic of a nonnumerical problem is that there is no algorithm or standard procedure for solving it. More precisely, the algorithms needed for solution are very complex --- if they are known to exist at all. In addition, there is a need for a unit of data larger than a single number, and the data must be able to change in both structure and content as the problem is solved.

A solution to this problem has been found in the list. A list may be defined as an ordered set of tokens, connected in a structure such that at least one successor to each token can be identified. [35] In some list structures, predecessors of tokens can be similarly identified. A simple example of a list is an ordered set of integers. The successor of one integer is the "next larger" integer. A more complex example is illustrated in Figure 1. In this example (taken from [35]) the address of each word in the list is stored in the word that precedes it in the list. Each word in the list may contain two addresses; one is the address of the item that belongs at that point, and the other is the address of the next word in the list. The latter address is the pointer address. The numbers outside the boxes are location addresses for the boxes. The list tokens indicated in this example comprise the symbolic expression:

(NAME-LIST/WORD)+ABC

Note that the order of the tokens is given by the pointer addresses; the tokens need not be listed in memory in sequence. The pointer address 00000 signals the end of the list.

With a list structure, the insertion or deletion of tokens is relatively simple. For example, to insert the subexpression +DIGIT immediately after NAME in the expression in Figure 1, the pointers are modified as shown in Figure 2. Two words were added to the list; only one pointer already in the list was changed (from 02205 to 02221), and everything else in the list was unchanged. Two tokens were added to the block at 04000. Deletion of tokens is also

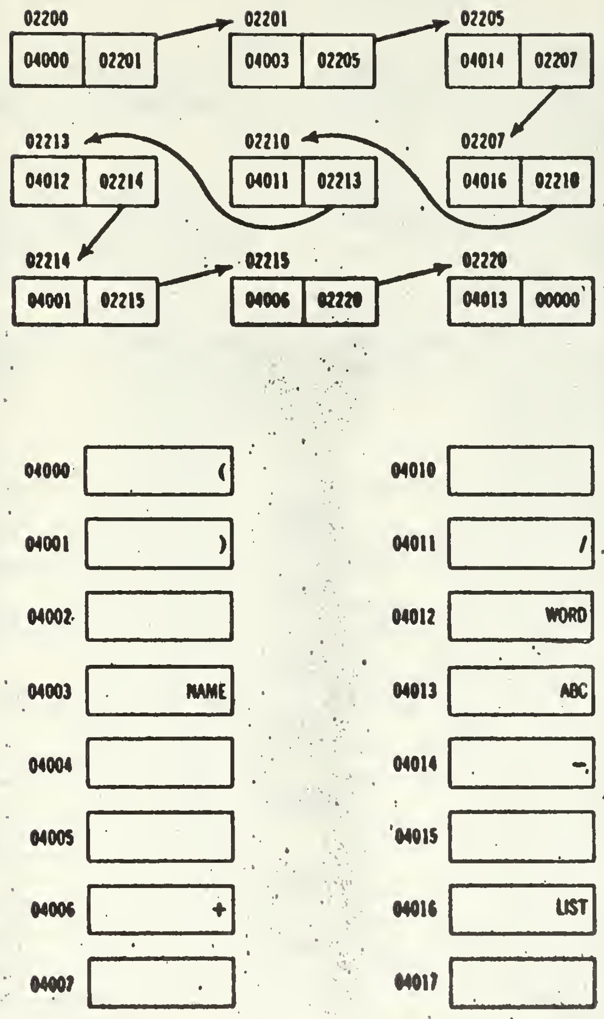
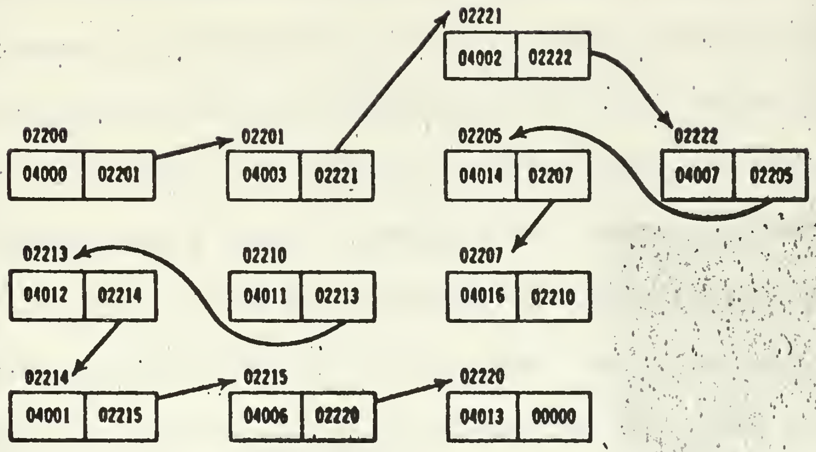


Figure 1.

Sample of a list corresponding to:

(NAME-LIST/WORD)+ABC



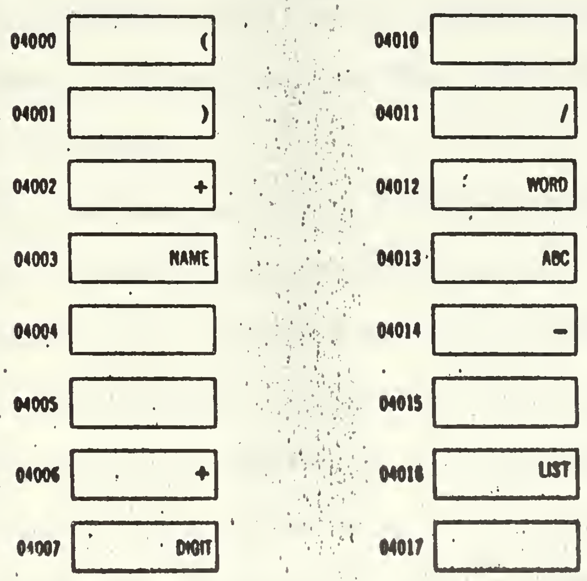


Figure 2.

Sample of the modified list corresponding to:

(NAME+DIGIT-LIST/WORD) + ABC

simple; pointers are modified, and nothing else is done. There will be extraneous information in words that were once part of the list, but since pointers bypass these words, no reference is made to them.

As lists are processed, many are replaced by others or are for some reason of no further use. If the space taken by such lists is never again used, the available space within the computer may eventually be depleted. The words no longer needed are scattered all over memory, and a means is required for determining where they are. This is accomplished by forming a list of available space (LAS). In this list each word points to a successor and to an available word. This list has the same structure as the list described. If the scheme is to function properly, each list operation must supply to the LAS those words made free by the operation. Thus at any time all free space is linked into one list. The programmer is free of worry over the space.

Several languages have been developed during the past few years which use lists and list structures as a fundamental working unit. A brief review of this work illustrates some of the applications of list processing languages and the various ways in which list processing can be introduced into programming.

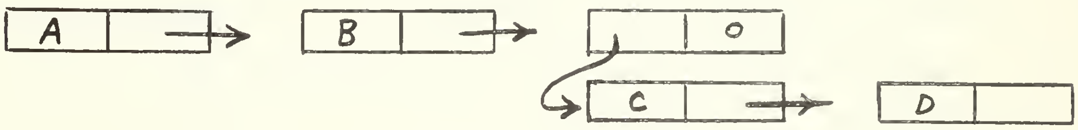
The work of Newell, Shaw, and Simon began in late 1954, triggered by the pioneering work of Selfridge and Dineen on a program for recognizing visual patterns. [4] [34] They first worked on chess and then switched to the task of proving theorems in the propositional calculus of Whitehead and Russell. [26] They originally devised some languages that were tied closely to subject

matter---a "chess language" and a "logic language". These languages, collectively called IPL-I (Information Processing Language-I), although designed as pseudo-codes, never reached the coded stage (see [25] for description). [24] [39] [27] The IPL series, I through VI, was developed for use with symbolic logic programs and hueristic programs such as chess programs, programs for use in management (e.g. assigning tasks to work stations for an assembly line) and a program called GPS, for General Problem Solver, which was an effort to simulate human behavior.

H. Gelernter has developed a program for proving theorems in plane geometry. [9] [11] To do this he and his colleagues developed a list processing adjunct to FORTRAN for the 704 called FLPL for FORTRAN List Processing Language. [10] This was done by adding a series of subroutines to the FORTRAN system. FLPL uses lists only for data, since it uses FORTRAN produced machine code for routines.

J. McCarthy has developed another list language for the 704 called LISP, for List Processor. [22] This one, like IPL-V, uses lists for both routines and data. Externally, LISP uses a horizontal notation in which list structures are represented with the aid of parentheses. The identity of parenthetical notation and list structures can be seen from the following figure:

(A,B, (C,D))



Part of the reason for the development of LISP is McCarthy's work with M. Minsky on heuristic programs. There has also been some work done on analytic differentiation and integration of elementary functions in LISP.

Another list processing language which was designed to retain the advantages of the preceding languages while simplifying machine processing of lists is called a <u>Threaded List</u>. It was developed by A. J. Perlis and Charles Thornton. [29] A threaded list is a list structure in which the last element of each list specifies the location of the head of the list of which it is the terminal member. The advantage of this structure is that it permits the definition of various modes for sequencing through lists without requiring the use of the usual push-down lists for retaining sequencing information. This corresponds, in the representation of programs by list structures, to the representation of all computable functions iteratively rather than recursively.

Weizenbaum developed the KLS (Knotted List Structure) language while considering the relationship between computer languages and computer organizations, and the hardware-software tradeoffs which this relationship makes possible. [39] Later he developed SLIP (Symetric List Processor) which is a descendent of FLPL, IPL-V, Threaded Lists, and his own KLS. In this system each list cell carries both a forward and a backward link as well as a datum (token of information). It is symetric in the sense that its lists do not have a preferred orientation. Any operation which can be carried out on the top of a list can just as easily be carried out

on the bottom. It is a language system designed to be imbedded in a higher order language capable of calling machine language subroutines (like FLPL). [40]

The Philco Company has implemented the IPL-V language on their 2000 computer as a set of macro-operations, subroutines and conventions supplementing TAC (Translator-Assembler-Compiler, the assambly language for the 2000). These macros, subroutines, and conventions will be referred to as TALL (TAC List Language). TALL uses the loading facilities of TAC, the IPL-V primitive processes, and a set of subroutines performing the work of the interpreter. The macros aid in the translation from IPL-V to TAC. The macros and the primitive processes can be placed on the TAC subroutine library tape and called in as required during assembly.

The implementation of IPL-V in this fashion has several advantages: (1) the time required to get a basis IPL-V system running on the 2000 is only three man-weeks; (2) symbolic machine language instructions can easily be inserted into TALL programs; (3) IPL-V statements can be used in conjunction with FORTRAN statements or JOVIAL statements; and (4) no additional work is required to make TALL compatible with any monitor system for the 2000. [5]

Compilers for one or more list processing languages exist today for every major general purpose computer.

In recognition of the fact that not all problems are purely of the symbol manipulation type, it might be well to mention a technique for symbol manipulation <u>and</u> numerical calculation. Ross has developed a type of generic structure which he calls a <u>plex</u>. [31]

There exist problems which are purely numerical, thereby not requiring symbol manipulation techniques, and there also exist problems which are purely symbolic, thereby not requiring efficient handling of data for computation. But there is a much larger class of problems in which both symbolic manipulation and numerical calculation are required and are inextricably intertwined. The plex structure is designed to handle this latter type. A claim is made that this structure is more powerful than the list structure (it includes lists as a subcase) and appears to be better suited for the concise representation of the complex interrelations of elements which constitute a "problem". [31] Note: The word plex is an abbreviation of the word plexus: "An interwoven combination of parts of a structure; a network." ----Webster.

3. Information storage and retrieval.

One important subset of the set of non-numeric applications of the digital computer is information storage and retrieval. The most recent and profitable improvements in the capabilities of digital computers have been the result of software innovations, i. e. assembly programs, subroutines, compilers or translators, etc. However, as the result of work in the symbol manipulation area, certain hardware specifications have been shown desireable. These may be classified into two main groups, 1) those which apply to the modern random access magnetic core type memory, and 2) those which apply to the new associative type memories. Typical of the references of the former are [41] and [23], and of the latter [30] and [28]. It is not the intent of this paper to become involved in detailed hardware comparisons. On the other hand, most of the discussion that follows is concerned with software solutions to problems related to storage and manipulation of structured information within the magnetic core, random access, type computers (the most common in-use general purpose computers). A word of warning to the reader is appropriate at this point. The literature on this subject often uses the term "associative memory" without distinction between the true associative memories and associative memories implemented in present day addressable memories by simulation techniques, e. g. see [17]

Most of what has been written regarding information storage and retrieval starts with the very convenient assumption that the information has previously been formatted, keyed, and stored. [32]

This does not mean, however, that the subject of automatic determination of content of documents has been completely neglected. There have been several attacks on the problem.

A well known method involves a frequency count of words in the text; those words which occur most frequently are judged to be the most significant and are used as indicators of the content of the document. A promising variant to this method has been proposed which utilizes only the initial occurrences of nouns of the text to detect document content. The results indicate that the proposed method may be useful for both the automatic indexing of documents and the production of automatic abstracts. [13]

A completely different approach to automatic indexing and classification is the use of bibliographic references provided with the document as quasi-index terms to improve indexing methods. An outstanding advantage of this method is that processing the full text of the document is not required. Although the formats of bibliographic references may vary in different documents, there has been some success at using a computer program to put these various forms into a standard form, so that they may then be processed by another program.

Another class of techniques to aid in the automatic determination of document content utilizes a syntactic analysis of some of the sentences of the document. In a retrieval system of this type there are numerous alternatives available in the degree and thoroughness of analysis.

The use of the semantic values of the words of documents has been used by several researchers in the area of data retrieval. There are difficulties in this approach as might be expected when different authors naturally use different words to express similar ideas. These semantic relations may also be supported by a syntactic analysis as mentioned above.

Another area of information storage and retrieval which has been thoroughly investigated is the <u>organization</u> of structured information in a computer and the effects of various allocations systems upon search strategies and matching procedures. Three basic formats --- vector, tree, and graph --- are encountered in the literature for the representation of structured information. Although the three forms are closely related, it is often convenient to consider them separately when analyzing characteristics of systems associated with them.

During the past several years, techniques for the evaluation of information storage and retrieval systems have been developed. Swets has reviewed ten different measures for evaluation and proposes another based on statistical decision theory. [38]

The various measures reviewed have much in common. Eight of them evaluate only the effectiveness (accuracy, sensitivity, discrimination) of a retrieval system and are derived completely, in one way or another, from the two-by-two contingency table of pertinence and retrieval represented in Fig. 3. The other three measures assess efficiency as well as effectiveness by including such performance factors as time, convenience, operating cost,

and product form. The questions of effectiveness and efficiency are difficult to answer and care must be taken in any comparative analysis. [38] [20] [21]

	P	$\bar{P}$	
R	a V_1	b K_1	$a+b$
$\bar{R}$	c K_2	d V_2	$c+d$
	$a+c$	$b+d$	$a+b+c+d$

Figure 3.

The Two-by-Two contingency table of pertinence and retrieval. P and $\bar{P}$ denote, respectively, pertinent and nonpertinent items; R and $\bar{R}$ denote, respectively, retrieved and unretrieved items; a, b, c, and d represent the simple or weighted frequencies of occurrence of the four conjunctions; V_1 is the value of retrieving a pertinent item; V_2 is the value of not retrieving a nonpertinent item; K_1 is the cost of retrieving a nonpertinent item; and K_2 is the cost of failing to retrieve a pertinent item. [38]

4. File processing operations.

One of the major considerations when deciding upon a particular storage allocation scheme for use in an information processing system is its effectiveness and/or efficiency. In order to make meaningful comparisons between various schemes considered, it is necessary to use parameters and file processing operations which are common to them all, insofar as this is possible. Iverson has developed a programming language and a notation for the description of programs. [16] Part of his notation is used throughout the remainder of this paper. See Table 1 for a summary of parameters and program symbols.

Consider a system where the major subdivision of information is a file. Let each file be further subdivided into records, and the records into items. In order to realize the ability to manipulate the files as desired, some means must be established to recognize units of information down through the smallest, i. e. the item. Each item, therefore, must have associated with it a tag or key, or descriptor which is unique. In fact, it is useful to define the item as having two parts, 1) the <u>key</u> which is that part which distinguishes the item from all other items, and 2) the <u>function</u> which is that part which is not the key. [37]

Each key corresponding to an item of the file is assumed to be an h-tuple, viz.,

$\underline{s} \in S_1 \times S_2 \times \ldots \times S_h$ where $\underline{s}$ = an argument key
S_i = <u>ith</u> set

and each of the sets, S_i, is considered to have m mebers or characters. Thus there are m^h distinct keys possible, e. g. if we restrict the keys to any six alphanumerics (a 6-tuple), then:

$$h = 6$$

$$m = 36 = 26 \text{ letters} + 10 \text{ digits}$$

$$m^h = 36^6 \cong 2 \times 10^9$$

The file is assumed to contain n items, all distinct. All items have the same probability of being selected.

If a parameter d is chosen so that $d^h = n$, then d/m is a measure of the frequency of occurrence of each character.

The physical memory in which the file is stored is assumed to contain 2^a words, so that "a" bits are required to address any word uniquely. The representation of each character is assumed to take b bits, where $m \leq 2^b$. Thus hb bits are required to represent an h-tuple $\underline{s}$. Therefore, in our example, b = 6 and hb = 36.

Two types of criteria are used to compare the storage allocation methods: the storage capacity required and the time required to perform a particular file-processing operation.

Three measures of storage capacity are used: 1) the number, w , of bits per memory word required to represent both an item and any allocation information associated with that item (w = number of bits per memory word if the word length is variable, and equals some integral multiple of a standard word if the word length is fixed), 2) the total number of words necessary to represent the file in a particular allocation, and 3) the total number of bits necessary.

It is convenient to normalize the latter two measures: [36]

$$r(\text{words}) = \frac{\text{total number of words to represent the file and associated allocation data}}{\text{number of words to represent the file}}$$

$$r(\text{bits}) = \frac{\text{total number of bits to represent the file and associated allocation data}}{\text{number of bits to represent the file}}$$

By the assumptions of item and file size the denominators of r(words) and r(bits) are n and $nb(h + e)$, respectively. However, computations are simplified if nbh is used as the number of bits to represent the file.

In computing the storage capacities required, the locations needed to store the programs and intermediate operating data are ignored. This is justifiable for the programs (q.v.) for similar searches in different allocations all have approximately the same number of lines and use about the same number of parameters. Moreover, the fraction of total storage used for the programs is much less than that used for the file in applications where these schemes are necessary.

Typical file-processing operations with respect to which time comparisons are made are: [36]

search for match
search for multiple match
search for smallest item
search for item next smaller than a given item
add an item to the file
delete an item from the file

Iteration of either "smaller" search can sort the file. Iteration of the add-item operation may be construed as a recursive procedure for constructing the allocation. Thus if the time to construct the allocation is required, one computes

$$\sum_{i=1}^{n} t_i$$

where t_i is the add-item time for a file of i items.

Two items, $\underline{a}$ and $\underline{b}$, are said to <u>match</u> ($\underline{a} = \underline{b}$) if and only if, for all i, $\underline{a}_i$ is the same character as $\underline{b}_i$.

Two items, $\underline{e}$ and $\underline{b}$, are said to <u>match with respect to $\underline{u}$</u> (wrt $\underline{u}$), where $\underline{u}$ is a logical vector of dimension h, if and only if $\underline{u}/\underline{a} = \underline{u}/\underline{b}$; that is, $a_i = b_i$ when $\underline{u}_i = 1$ (here we are using Iverson's notation. In particular, $\underline{u}/\underline{a}$ is his "compression" operation). [16] $\underline{u}$ is called a <u>mask</u>. If the mask $\underline{u}$ is not explicitly mentioned, it is assumed to be a vector or all 1's.

An order relation is defined on S as follows: For those S_i that are ordered, let " $<$ " be the relation "precedes". Roughly speaking, $\underline{a} < \underline{b}$, if, when the unordered sets are ignored and the ordered sets are considered to be nonnegative integers, the "number" $\underline{a}$ is less than the "number" $\underline{b}$.

The restriction of the ordering with respect to a mask $\underline{u}$ is defined: $\underline{a} < \underline{b}$ wrt $\underline{u}$ if and only if $\underline{u}/\underline{a} < \underline{u}/\underline{b}$.

Those items $\underline{s}$ of file F which are such that, for all $\underline{t}$ in F, $\underline{s} < \underline{t}$ wrt $\underline{u}$, are the <u>smallest items of the file wrt $\underline{u}$</u>.

The items next smaller than $\underline{s}$ wrt $\underline{u}$ are those items $\underline{t}$ in F such that $\underline{t} < \underline{s}$ wrt $\underline{u}$ and for which there is no $\underline{r}$ in F such that $\underline{t} < \underline{r} < \underline{s}$ wrt $\underline{u}$.

The definition of the ordering relation " $<$ " requires that the elements of the h-tuple are weighted by the order in which they occur in the h-tuple. This may be generalized to utilize different orderings as specified by some permutation vector $\underline{p}$. The restricted ordering with respect to mask $\underline{u}$ may still be used, so that $\underline{a} < \underline{b}$ wrt $\underline{u}$ wrt $\underline{p}$ if and only if $\underline{u}/\underline{a} < \underline{u}/\underline{b}$ wrt $\underline{u}/\underline{p}$ (The expression $\underline{x} < \underline{y}$ wrt $\underline{z}$ is not ambiguous because a given $\underline{z}$ cannot be both a permutation vector and a mask vector except in the trivial case $\underline{z} = (1)$). If the permutation $\underline{p}$ is not explicitly mentioned, it is assumed to be the identity permutation.

A match search of file F for item $\underline{x}$ wrt $\underline{u}$ is an operation which identifies that item $\underline{s}$ of F such that $\underline{x} = \underline{s}$ wrt $\underline{u}$. To qualify as a match search $\underline{u}$ must be so chosen that the search identifies exactly no or one item of the file. If $\underline{u}$ is such that several items of the file may be identified, the search is called a multiple-match search. In general a multiple-match search is more complex than a match search so that it is assumed that the user of the file knows when to expect a unique result and when he will be content with a single response from many possibilities.

As an example illustrating the possible uses of the suggested searches, consider the file to be a personnel record of a large corporation. [36] In such a file each item would correspond to one employee and might be categorized as:

Name

Social security number

Department number

Job code

Salary

Hence, each item is a 5-tuple. Note that in this example the sets of the file are:

$S_1 \approx$ the set of employees' names

$S_2 \approx$ the set of employees' social security numbers

$S_3 \approx$ the set of departments

$S_4 \approx$ the set of job codes

$S_5 \approx$ the set of salaries

We have obviously departed from our original example where the sets S_i were the set of alphanumeric characters. This is done to make the explanation of searches more easily seen.

Probably the most common search of such a file would be a match search to locate the item for a particular individual: e.g.

$\underline{u} = (1,0,0,0,0)$

$\underline{s}$ = (Smith John A,-,-,-,-)

To locate all the secretaries in department 474, a multiple-match search would be conducted with

$\underline{u} = (0,0,1,1,0)$

$\underline{s}$ = (-,-,474,secry,-)

The payroll department might desire an alphabetical listing of all employees by departments and salary; that is, sort the file

with major category "salary", then "department number", then alphabetically by "name". To do this, a next larger search is iterated with

$\underline{u} = (1,0,1,0,1)$

$\underline{p} = (3,-,2,-,1)$

$\underline{s}$ = last item retrieved

and

initial $\underline{s} = (A,-,0,-,0)$.

5. Storage allocation systems and their relative merits.

When reduced to basic considerations, all information storage systems for addressable storage within a digital computer memory use one or more applications of four basic types, i. e. random, ordered, computable, or chained. Each has its own advantages and disadvantages. As we shall see later, more complex systems which are combinations of these basic systems may be devised which allow optimization for particular applications.

In this part the four simple storage allocation systems mentioned above are examined. For each, the description of the allocation, discussion of various searches, programs for the searches, and calculations of search times and storage requirements are given. The treatment here has been adapted from Sussenguth with minor modifications. [36]

The programs delineating some of the search descriptions are intended to convey a feeling for the type and amount of processing required, rather than being exact programs which cover all contingencies. Indeed, the details of initialization and special cases (e.g., multiple responses to a smaller search) are frequently omitted or condensed to a single program step. In contrast to this supression of detail, in the main iterative loop of the program, the step of retrieving an item from memory

$$\underline{a} \leftarrow \underline{M}^{i}$$

is explicitly shown. It is this operation that is assumed to be the most time-consuming, and it is the number of times that this

step is executed that is reflected in the search times which are calculated. If the operations of initializing, indexing, comparing, etc. are not carried out in parallel with the retrieval operation, or their execution times are not negligible with respect to it, the program will give an idea of how much the memory-access time must be increased to reflect a proper item time.

When a program terminates at a normal exit (error exits are marked with an asterisk), the item satisfying the search is contained in the accumulator a. For those searches which may be satisfied by several items, the line "process a" is used to indicate that the present contents of the accumulator satisfy the search; after this line has been executed, the search continues until it terminates at an exit.

The programs are all written in the Iverson language. Table I, p. vii gives the legend which is used for all programs.

The measure of time required to perform an operation is the number of items that must be "handled" to achieve the desired objective. "Handling an item," an intentionally vague phrase, may be construed as extracting an item from memory and transferring it to the processing unit in which a computation or decision is performed based on that item. In general, many items will be handled to produce the file operation. By defining an item time in this way, any explicit real time measure (i.e., number of seconds) is eliminated but, in so doing, one must avoid the impression that item times for different operations and different allocation schemes are the same with respect to real time. This

is, of course, not true in general. To give some feeling for the real time measure of an item time, simple skeleton programs for the basic file operations will be given for most storage allocations. The program steps may then be weighted in terms of operating times of a particular computer. The search times which are computed, then, are just a count of the number of file items which must be retrieved from memory to perform the desired task.

If the program and intermediate operating data are stored in the same memory as the file, the real time value of an item should include the number of retrievals of both instructions and data required to perform one iteration of the program. If, however, the program and intermediate parameters are stored in an auxiliary high-speed (relative to the file storage) memory, it may be safe to ignore their effect. In this latter case search times for different allocations and different operations may be directly compared.

The search times and storage ratios computed for each allocation are summarized in Table 2,pp 47,48.There are three entries for each search: the maximum, minimum, and expected number of item times to complete the search. Frequently the entries are approximations of the expressions which have been derived. When applicable, the $\underline{u}$ or $\underline{p}$ for which the entries are calculated are shown; for most other $\underline{u}$ and $\underline{p}$ the search reduces to the corresponding search of the random allocation. Table 3 repeats the average values of Table 2 but with typical values chosen for the file parameters.

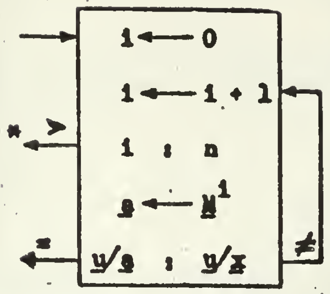
A. Random

The simplest storage allocation system is that in which the items are stored in a random fashion in consecutive memory locations. Thus, there is no structure to this organization and it probably does not deserve being classified as an allocation system. It is important to consider it, however, because it serves as a basis of comparison for other systems, the cost of a particular allocation system being considerably less than the costs involved here if the other system is at all efficient. Moreover, many allocations are designed to achieve maximum efficiency for one or two particular operations and may be considered to be random structures with respect to other operations. An important example of this is the smaller searches performed with respect to a permutation other than the identity permutation.

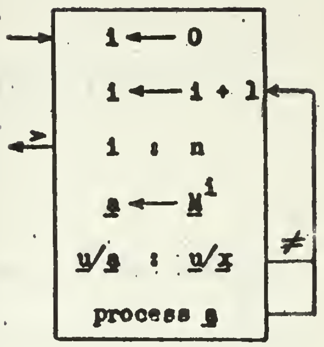
Searching for a match in the random structure consists in examining the items one by one until the desired item is found (program A1); hence, the maximum search time occurs when the matching item is the last one of the file and thus requires n item times, the minimum time is 1 when the first item matches, and the expected time is n/2.

For a multiple-match, $\underline{x}$ wrt $\underline{u}$, all items must be tested (program A2) as there is no a priori way of determining how many items satisfy the criterion.

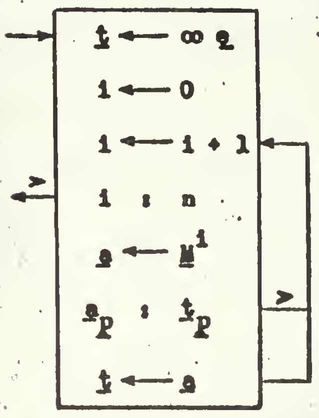
Similarly, it is necessary to test all items to determine the smallest item or the item next smaller than $\underline{x}$ wrt $\underline{u}$ wrt $\underline{p}$ (program A3 and A4).



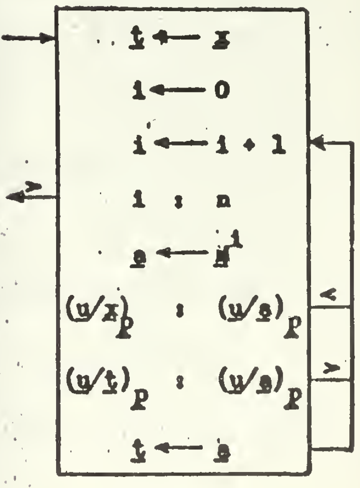
Program A1
Match search, x wrt u



Program A2
Multiple-match search, x wrt u



Program A3
Smallest search wrt p



Program A4
Search for next smaller
than x wrt u wrt p

[illegible]

[illegible]

[illegible]

[illegible]

Adding an item to the file is particularly simple, for it is inserted at the end of the file. Deleting an item is only slightly more complicated: the item is located and replaced by the last item on the list (replacement is necessary to preserve the solid nature of the storage.) Thus, deletion item times are just the match search times plus one.

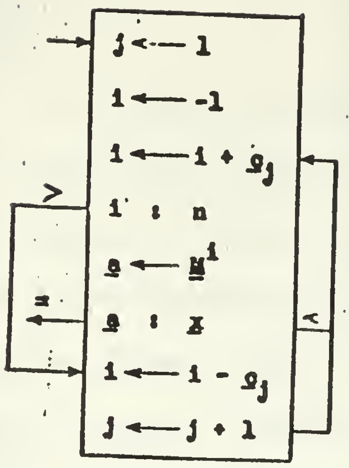
As no additional structural information is required for this allocation scheme, r = 1.

B. Ordered

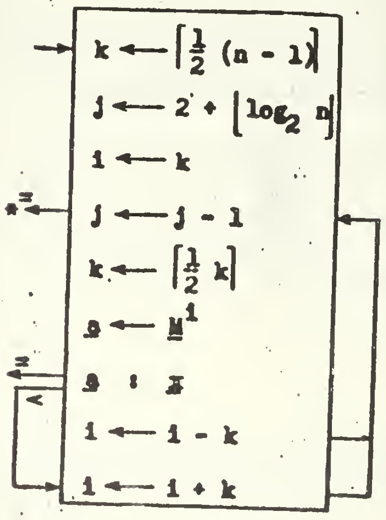
In an ordered file the items have been sorted by some external means into an increasing order with respect to some $\underline{u}$ and are stored in consecutive locations of the memory.

A dictionary provides a simple example of such a file. To locate a specified word the thumb index is used to find a location near the desired entry, then the catchwords at the top of the pages are used to narrow the number of candidate items to a few dozen, and finally an entry-by-entry search is made to find the required entry. Thus, in general, the search for a match in an ordered file may be conducted by successively finer scans of the file, the first scan determining a "general area," the succeeding scans reducing the number of candidates until an exhaustive search is performed. Program B1 shows such a search in which the elements of the vector $\underline{c}$ determine the coarseness of the scans.

Obviously, the minimum search time is 1, wherein the first item tested matches. The maximum number of items which may be


$j \leftarrow 1$
$i \leftarrow -1$
$i \leftarrow i + \underline{\omega}_j$
$>$
$i : n$
$\underline{s} \leftarrow \underline{M}^i$
$=$
$\underline{s} : \underline{x}$
$<$
$i \leftarrow i - \underline{\omega}_j$
$j \leftarrow j + 1$

Program B1
Dictionary match search for $\underline{x}$


$k \leftarrow \lceil \frac{1}{2}(n-1) \rceil$
$j \leftarrow 2 + \lfloor \log_2 n \rfloor$
$i \leftarrow k$
$* =$
$j \leftarrow j - 1$
$k \leftarrow \lceil \frac{1}{2} k \rceil$
$\underline{s} \leftarrow \underline{M}^i$
$=$
$\underline{s} : \underline{x}$
$<$
$i \leftarrow i - k$
$i \leftarrow i + k$

Program B2
Binary match search for $\underline{x}$

tested in the first scan is $\lfloor n/\underline{c}_1 \rfloor = g_1$, the maximum on the second scan $\lfloor \underline{c}_1/\underline{c}_2 \rfloor = g_2$, etc., so that the overall maximum search time is

$$\sum_i g_i .$$

If $n = d^h$ and $\underline{c} = (d^{h-1}, d^{h-2}, \ldots, 1)$, then the maximum time is hd. An estimate of the expected search time is found by assuming only one half the maximum number of items are tested on each scan.

Another search technique for the ordered file, closely related to the "dictionary" search, is the binary search in which, at each access to the file, the number of possible candidate items is successively halved until either the retrieved item matches or only one candidate remains (program B2). Again the minimum search time is 1. The maximum time occurs when the last item tested matches; this time is $\lceil \log_2(n+1) \rceil$. To determine the expected search time, t, notice that on the initial entry to the file only one item may be tested, on the second entry either of two entries is tested, one of four on the third, and one of 2^{i-1} items is tested on the ith entry. Thus,

$$t = \sum \text{(probability of selection of ith item) X (number of entries to reach the ith item)}$$

$$= \sum_{i=1}^{g-1} \frac{i 2^{i-1}}{n} + \frac{g}{n}(n+1-2^{g-1}) = \frac{1}{n}(g2^{g-1} - 2^g + 1) + g + \frac{g}{n} - \frac{g2^{g-1}}{n}$$

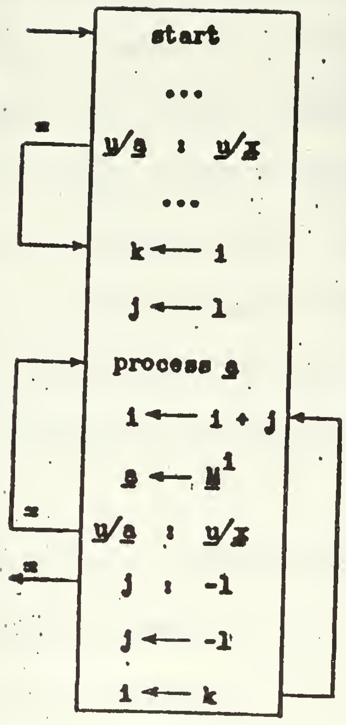
$$= g - \frac{1}{n}(2^g - g - 1)$$

where $g = \lceil \log_2(n + 1) \rceil$.[1] If $n \doteq 2^g$ and $2^g \gg g$, the expected search time becomes $g - 1$. Note that this g is not the same factor as used later in the discussion of trees.

The search for multiple matches of $\underline{u}/\underline{x}$ is not difficult if the mask vector is a prefix of the vector on which the items are ordered: A match search is conducted to find any item satisfying $\underline{u}/\underline{x}$; successive neighboring items "below" this item are retrieved until an item fails to satisfy the $\underline{u}/\underline{x}$ match; similarly, items "above" the initial item are retrieved (program B3). The various search times are thus the appropriate match search time plus the number of items satisfying the match plus two (two items retrieved do not satisfy $\underline{u}/\underline{x}$). If $\underline{u}$ is not a prefix vector, the search is essentially that of the random structure although modifications may be made to account for leading 1's in $\underline{u}$.

Similarly, the search for the smallest item and the item next smaller than a given item become trivial if the search is with respect to the $\underline{u}$ on which the items are ordered. If this is not the case, the file must be considered just a random listing.

[1] The closed forms of all nontrivial summations are given in the appendix.



Program B3

Multiple-match search
for x wrt prefix y
(B3 is B1 or B2 with a minor modification
and augmented by nine lines.)

To add an item to an ordered file, a match search is performed to determine its proper position and then other items are moved to allow for its insertion. If the location of the top of the file is fixed, i. e., not subject to modification, all of the items greater than the new item must be moved down to allow for its insertion. Thus, at most n items must be moved (new item is the smallest item), at least no items must be moved (new item is the greatest), and an average of n/2 items must be moved. If neither the top nor the bottom of the file is fixed, the maximum number of items which must be moved is n/2 and the average number of n/4. Hence, to obtain approximate times for the addition of an item, the respective match search times are added to either n, n/2, n/4, or 0, as appropriate. Deletion times are determined in an analogous manner.

As only the file items themselves are stored, w = hb and r = 1.

C. Computable

A storage allocation of a file in which the location of an item is a function of a subset of its elements, i. e.,

$$\text{Location}\ (\underline{s}) = f(\underline{u}/\underline{s})$$

is called a computable allocation. Frequently, f is a linear function although it generally need not be.

A common example of a computable allocation is the storage of a matrix $\underline{A}$ in which the location of the coefficient $\underline{A}_{ij}$ is a linear function of i and j:

$$\text{location}(A_{ij}) = \alpha i + \beta j + \gamma$$

The common practice is, of course, to store only the coefficient $\underline{A}_{ij}$, not the 3-tuple $(i,j,\underline{A}_{ij})$, at the designated location.

This practice of not physically storing in memory those elements of the item used as the arguments of f is advantageous in that a shorter word length may be used. This gain may be compensated by a loss in efficiency in some search, however. For example, if matrix $\underline{A}$ is searched to find its smallest coefficient, it may be difficult to determine the associated indices i and j, if they are not explicitly stored. If such searches are infrequent, this disadvantage is insignificant, and then it is necessary to store only the $\bar{\underline{u}}/\underline{s}$ portion of each item s. Thus, the peculiarity of r(bits) being less than unity arises: indeed, if $\bar{\underline{u}}/\underline{s}$ has z elements,

$$r(\text{bits}) = \frac{nbz}{nbh} = \frac{z}{h} .$$

The use of the word "search" in a computable allocation seems out of place, for an item is located by simply computing $f(\underline{u}/\underline{s})$. Nevertheless, the time to compute f and retrieve the associated item shall be considered to be the match-search item time. This time has been designated f rather than 1 in Table 2 to distinguish computation time from retrieval time; the relative magnitude of f with respect to 1 is not restricted. (In the matrix example, it may be necessary to retrieve α, β, and γ from the same memory as the items themselves, wherein f would be about 3; or α, β, and γ might be stored in a high-speed memory especially reserved for such constants, wherein f might be much less than 1.)

A multiple-match with respect to $\underline{v}$ involves computing f for all values $\underline{v}/\underline{u}/S$. (In the matrix example, retrieving all the items of a particular row is a multiple-match search.) The multiple-match time is a computation times, where s is the product of the orders of the sets $\overline{\underline{v}}/\underline{u}/S$.

Searches for the smallest item and the item next smaller than a given item also involve one computation.

Notice that the preceding arguments have been based on the premise that the search arguments are included in the domain of f, viz. in $\underline{u}/S$, and that the elements of the domain are all known. If this is not true, the computable allocations reduces to a random allocation. Moreover, the function f may be given arguments outside its domain and an ill-defined request may result in a legitimate item. In other words, the computation of the location should be a two-step process: It must first be ascertained if the argument $\underline{u}/\underline{s}$ is in the domain of f; then, if it is, $f(\underline{u}/\underline{s})$ is computed. For example, if $\underline{A}$ is an 8-by-8 matrix, and the value of $\underline{A}_{3,9}$ is requested, one might obtain the value of $\underline{A}_{4,1}$ without any error indication.

The addition or deletion of an item to a computable allocation is very difficult, because a completely new f must be constructed in most cases (possibly involving a reallocation of many items also). This may be circumvented when deleting an item by retaining f but inserting a dummy item in place of the one being deleted. If the dummy item is retrieved in some search thereafter, it is ignored.

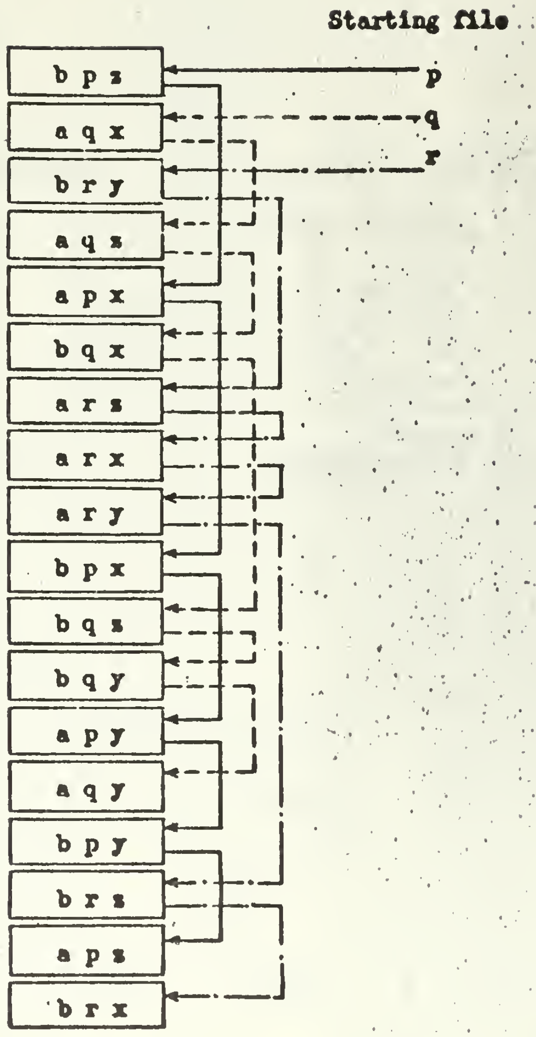
D. Chained

In a chained allocation with each item of the file is associated the addresses of other items of the file. Thus when one item is retrieved, information is supplied as to how to locate other items of the file. Chaining items together in this way eliminates the need for storing the file in consecutive memory locations, greatly simplifies the problems of adding or deleting items, and results in reasonable search times.

The chained allocation has search times comparable to the random allocation if the items chained together do not reflect any structure inherent in the file. If, however, items with common characters are chained together, the search times can be substantially reduced. This is accomplished by chaining together items with similar characters in a given character position. Thus if the chaining is within set S_i, i. e. the ith character position, and the set has m distinct elements, there will be m chains each of (average) length n/m. An example of a chained allocation is shown in Fig. 4.

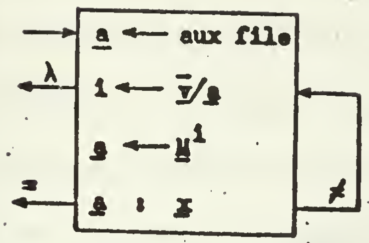
When searching for a match, then, that chain which agrees with the appropriate character of the argument is followed and each item retrieved is tested for the match condition by comparing it with the argument item (program D1). The maximum, minimum, and expected search times are n/m, 1, and n/2m respectively.

A similar procedure may be followed for multiple-match searches, but the entire chain must be traversed to insure all matching items are located (program D2).

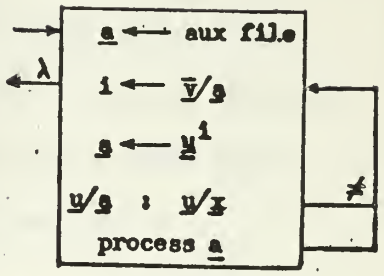


Schematic representation of a chained allocation.
File $F = S_1 \times S_2 \times S_3$ where $S_1 = \{a,b\}$, $S_2 = \{p,q,r\}$, $S_3 = \{x,y,z\}$.
The chaining is within S_2.

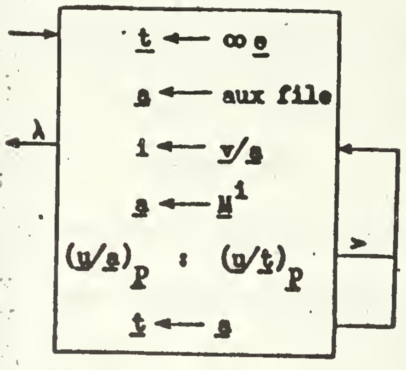
Fig. 4



Program D1
Match search for x



Program D2
Multiple-match search, x wrt u



Program D3
Smallest search wrt u wrt p

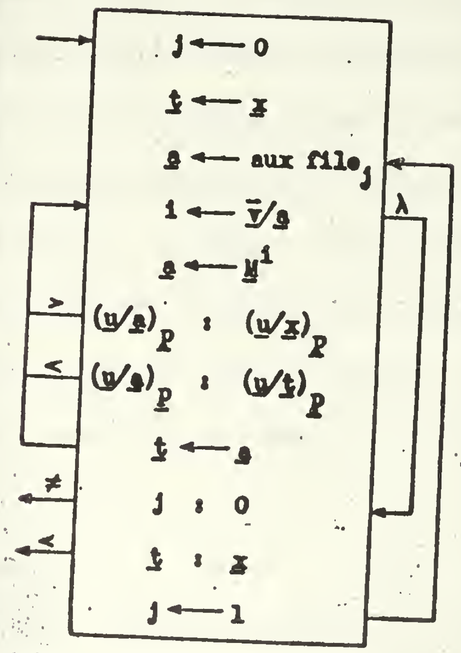
For these searches it is necessary that the element of $\underline{u}$ corresponding to the set on which the chaining is constructed is 1 (i.e. $\underline{u}_i = 1$ where the chaining is within S_i). To insure this, it may be necessary to have sets of chains on several different character positions so that there is at least one chain which can be followed for all $\underline{u}$ which may be of interest.

To search for the smallest item, the entire chain corresponding to the least character of the most significant character position is traced (i. e. follow the chain corresponding to the least character of set S_i where i is such that $\underline{u}_i = 1$ and for all $j \neq i$ $\underline{p}_i < \underline{p}_j$). (Program D3)

The search for the item next smaller than a given item $\underline{s}$ is similar, but the chain corresponding to the same character of the most significant character position is followed (i.e. follow the chain corresponding to $\underline{s}_i$ where i is defined as above). If it should happen that $\underline{s}$ is the least item in this chain, the chain corresponding to the character next less than $\underline{s}_i$ is followed and the search is for the largest element of that chain (program D4).

Thus, the search time for the smallest item is always n/m, and the time for the next smaller item depends on whether or not it is necessary to "back up" one character. As at most one back-up is required (because it is the character in the most significant position which is altered) and the probability that it is required is m/n (because there are n/m items in each chain), the maximum, minimum, and expected search times are $\frac{2n}{m}$, $\frac{n}{m}$ and $\left(1+\frac{m}{n}\right)\frac{n}{m}$

respectively.



Program D4
Search for next smaller than $\underline{x}$ wrt $\underline{u}$ wrt $\underline{p}$

To find the first item in each chain, an auxiliary file of starting addresses is required. This is a listing of each character of each position which is chained and the address of the first item of the corresponding chain. Such files may be random, ordered, or computable allocations as may be appropriate. The time for searching the auxiliary file, which has m items, has been ignored in the search times of Table 2.

Notice that the chained structure is similar to the computable structure in that some characters of each item need not be stored since their value is implicit in the chaining. Thus, if $\underline{v}$ is the logical vector with 1's corresponding to the sets which are chained, only the elements $\overline{\underline{v}}/\underline{s}$ need be physically stored. No loss of information is incurred in doing this if the last item in each chain is connected to the auxiliary file; thus the appropriate character value may be ascertained whenever an item is retrieved via another chain. Although eliding more than one character position may lengthen the search times considerably, r(bits) may be greatly reduced, especially if $a \ll b$. It is assumed no elision has occurred in computing item times.

To add an item to a chained file, the item is written into any available location and the chaining addresses modified. A simple way to modify the chain addresses is to consider the new item to be the first in each chain so that the starting list addresses point to the new item and the addresses previously in the starting list are those of the new item. Hence, if there are c character positions

which have chains, c+1 items must be handled to add a new one (c starting items plus the new item).

To delete an item it is necessary to locate it by a match search and then replace the chaining address of its predecessor by the chaining address of the item being deleted. The item must be located in each of the c chains; following just one chain is not sufficient for once an item is located, its successors in each chain are known, but not its predecessors. Hence, c match-search times are required to delete an item.

In a chained allocation each word must contain w = hb + ca bits, hb for the item and ca for c chain addresses. Since characters which are chained together may be elided, it is possible to reduce the bit count to (h - c)b + ca. In addition to the n words of the file, cm words are required to hold the starting addresses, so that

$$r(\text{words}) = 1 + \frac{cm}{n} \, .$$

Similarly it is found that

$$r(\text{bits}) = 1 + \frac{ca}{hb} + \frac{cm(a + b)}{n} .$$

E. Evaluation of Results

Before attempting to compare the relative efficiencies of two allocation systems, one must be certain that like quantities are being compared. One of the bases for comparison that has been selected is the item time, which essentially reflects the number of file items that must be processed to achieve the desired result. It does not, however, reflect a real time measure and, hence, the equality of two search times does not imply equality of actual processing time. Therefore, before comparing allocations on the basis of the data of Table 2, some consideration of the real time values of the item times must be made.

The second basis of comparison chosen was the storage capacity which the file, plus the additional allocation structure, required. Three measures of storage capacity were computed: storage ratios r(words) and r(bits) and the number of bits per word, w. While it is mathematically true that

$$r(\text{bits}) = r(\text{words}) \times w/bh$$

this equation may not be true in a practical sense depending on the characteristics of the storage medium and the data processor. Suppose the memory and processor have a fixed word length of N bits. If

$$N/2 < w < N$$

for a particular file and application, then r(bits) has no real significance because one full word must be used to hold the necessary allocation data, wasting N - w bits of each word. Bits are similarly wasted if $N < w$ and w is not an integral number of words. If,

[illegible]

however, $w = N/2$, and the computer has the ability to process half-words efficiently, two allocation items may be packed into one word. Conversely, if the computer has a variable field-length feature (and many computers do),[1] r(words) loses significance because each allocation item uses only as many bits as necessary.

Thus, in comparing allocations with respect to required storage capacity, the characteristics of a particular computer should be considered to determine whether r(bits) or r(words) reflects the storage efficiency more properly.

It is by no means clear that operating time and storage capacity constitute reasonable measures. Indeed it is easy to improve one at the expense of the other. For example, a random allocation requires essentially no maintenance, has both r(words) and r(bits) as small as possible, but has very long search times; at the other extreme, duplicating the file in all conceivably useful configurations is extremely wasteful of storage space but permits rapid searching. One function which overcomes this type of difficulty and has a reasonable physical interpretation is the product of search time and storage capacity. The product may be thought of as the cost of operating the system, for the storage capacity is a measure of the amount of equipment required and the search time is a measure of the amount of time the equipment is in use.

[1]E.g. IBM 7030, 7080, 705, 1401, 1620; RCA 501, 301; Univac 1107.

The nature of the user's problem is also an important and obvious factor in the determination of relative efficiency. If, for example, additions to or deletions from the file are rare, it is meaningless to use these times as a basis of comparison. If, however, the number of additions to the file is large, one might consider a composite allocation of a main ordered file and a subsidiary random file, which is searched if the item is not located in the main file; by merging the new items all at once into the main file when, say, the random file becomes large, an efficient system may be developed.

With these words of caution in mind, let us now summarize the results of the previous parts. Tables 2 and 3 will be of assistance.

The random allocation has no desirable qualities other than the case of adding items and, hence, finds little application except as a temporary auxiliary file, as noted above.

The ordered allocation, which is probably the most widely used allocation system, is quite efficient with respect to both storage and search time, for files with infrequent changes. If the rate of change is not low, the file may be padded with dummy items, so that the deletion of an item is accomplished by merely designating it as a dummy and the addition of an item will involve moving only a few items. If a dictionary-type search is used instead of the binary search, slight rearrangements of the proper order within the fine scan range may be tolerable, thereby permitting even few item movements when adding an item.

A computable allocation is also quite efficient in both storage and search time, but the file items must have very special properties to allow the construction of a reasonable function. Therefore, when a computable allocation is possible, it probably is not recognized as such, but rather, the file is treated as a mathematical entity (e.g. a matrix or vector). A computable system is frequently used within another system, e.g. as the coarsest can in a dictionary search. Often the function evaluation only involves the use of index registers and indirect addressing.

Chained structures appear rather wasteful with respect to both search time and number of bits required. However, the main use of such structures has been, not for searching for matching items, but for linking complex data or program chains which are frequently changed. Chained structures have found wide use in symbol-manipulating, game-playing, and theorem-proving programs.

		Random	Ordered (binary)	Ordered (dictionary)	Computable	Chained
Max	Search for Match	n	$g = \lceil \log_2(n+1) \rceil$	$g = \sum_i \left\lfloor \frac{c_{i-1}}{c_i} \right\rfloor$	f	$\frac{n}{m}$
Min		1	1	1	f	1
Avg		$\frac{n}{2}$	$g - 1$	$\frac{g}{2}$	f	$\frac{n}{2m}$
$\underline{u}$		(1)	(2)	(2)	(3)	(4)
Max	Search for Multiple Match	n	$g + s + 2$	$g + s + 2$	sf	$\frac{n}{m}$
Min		n	$s + 3$	$s + 3$	sf	$\frac{n}{m}$
Avg		n	$g + s + 1$	$\frac{g}{2} + s + 1$	sf	$\frac{n}{m}$
$\underline{u}$		(1)	(2)	(2)	(3)	(5)
$\underline{p}$		(1)	(2)	(2)	(1)	(1)
Max	Search for Smallest	n	1	1	f	$\frac{n}{m}$
Min		n	1	1	f	$\frac{n}{m}$
Avg		n	1	1	f	$\frac{n}{m}$
$\underline{u}$		(1)	(2)	(2)	(3)	(5)
$\underline{p}$		(1)	(2)	(2)	(1)	(1)
Max	Search for Next Smaller	n	$g + 1$	$g + 1$	f	$\frac{2n}{m}$
Min		n	2	2	f	$\frac{n}{m}$
Avg		n	g	$\frac{g}{2} + 1$	f	$\frac{n}{m} + 1$

Summary of Search Times and Storage Ratios

TABLE 2

(continued on next page)

		Random	Ordered (binary)	Ordered (dictionary)	Computable	Chained
Max	Add an Item	1	$g+n$ or $g+\frac{n}{2}$	$g+n$ or $g+\frac{n}{2}$	large	$c+1$
Min		1	g	g	large	$c+1$
Avg		1	$g+\frac{n}{2}-1$ or $g+\frac{n}{4}-1$	$\frac{g}{2}+\frac{n}{2}$ or $\frac{g}{2}+\frac{n}{4}$	large	$c+1$
Max	Delete an Item	$n-1$	$g+n$ or $g+\frac{n}{2}$	$g+n$ or $g+\frac{n}{2}$	large	$\frac{cn}{m}$
Min		2	g	g	f	c
Avg		$\frac{n+2}{2}$	$g+\frac{n}{2}+1$ or $g+\frac{n}{4}-1$	$\frac{g}{2}+\frac{n}{2}$ or $\frac{g}{2}+\frac{n}{4}$		$\frac{cn}{2m}$
Storage Ratio (words)		1	1	1	1	1
Storage Ratio (bits)		1	1	1	$\frac{z}{h}$	$1+\frac{ca}{bh}$
Bits per Word		bh	bh	bh	bz	bh + ca

Mask vector $\underline{u}$ restrictions

1. any mask
2. mask must be prefix of ordering vector
3. mask must be within domain of f
4. mask must correspond to at least one chain
5. mask must correspond to chain of most significant digit

Permutation vector $\underline{p}$ restrictions

1. any permutation
2. those items not being matched in normal order, i.e. $\overline{\underline{u}}/\underline{p}$ $\overline{\underline{u}}/\underline{i}$

TABLE 2 (continued)

	Random	Ordered dictionary	Ordered binary	Computable	Chained
Match	10000	28	14	2	333
Multiple match	20000	128	116	200	667
Smallest	20000	1	1	2	667
Next smaller	20000	29	15	2	668
Add item	1	5028	5014	large	4
Delete item	10001	5028	5014	large	1000
Bits per word	36	36	36	24	81
Storage ratio (words)	1	1	1	1	1
Storage ratio (bits)	1	1	1	1	2.25

Parameters
a = 15
b = 6
c = 3 n = 20000
f = 2 s = 100
	$\underline{c}$ = (1000, 50, 10, 1)	two implicit characters	

Comparative item times for the typical parameters shown.

TABLE 3.

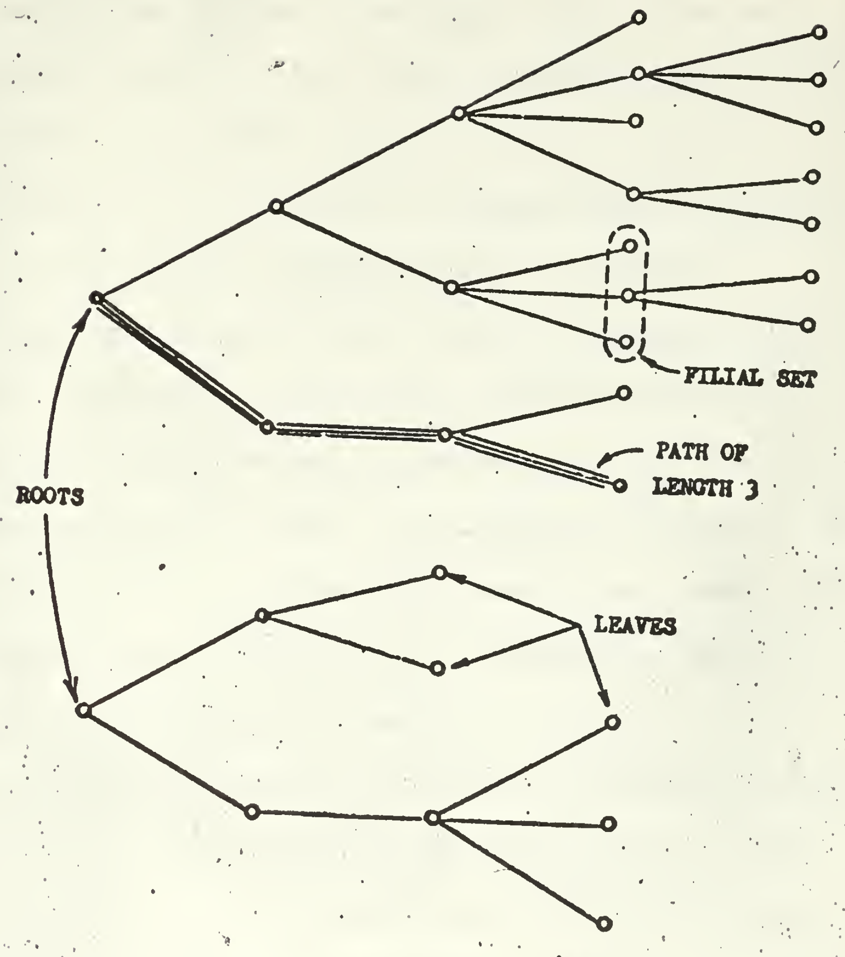
6. Representation by abstract trees.

The abstraction of a storage scheme which is a combination of two or more of the basic schemes discussed above is called a Tree. Organization of information, or keys to items of information, in the form of a tree allows utilization of certain advantages of the combined schemes with a minimum trade-off in terms of combination generated disadvantages. The method of combination and the variance of tree parameters allow the programmer an opportunity to maximize the desireable features of a given combination. This further allows scientific selection and provides a means to predict performance.

It is convenient to collect definitions used in the following discussion of tree structures. The terminology varies slightly with various authors but we will use the terms adopted by Iverson and Sussenguth. [16] [37] Several terms are illustrated in Figure 5.[1]

A graph comprises a set of nodes and a set of associations specified between pairs of nodes. If node i is associated with node j, the association is called a branch from the initial node i to the terminal node j. A path is a sequence of branches such that the terminal node of each branch coincides with the initial node of the succeeding branch. Node j is reachable from node i if there is a path from node i to node j. The number of branches in a path is the

[1]It may bother some readers that the pictorial representation of trees do not always branch upward as do natural trees. Such is not the case, however, and the literature abounds with trees "growing" every which way. Branching downward seems the most comfortable possibly as the result of the general familiarity with the organization chart heirarchical trees (with the boss on top).



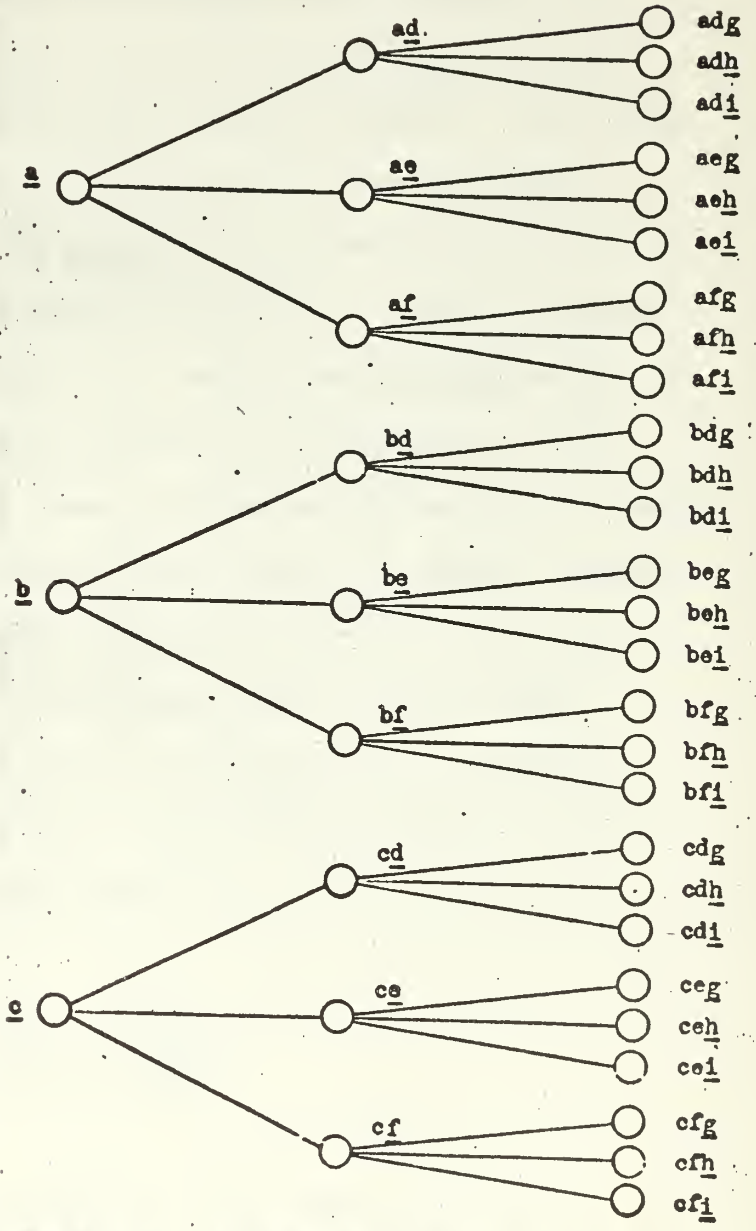
A Pictorial Representation of a Tree, Illustrating Some of the Terminology

Figure 5

length of the path. A circuit is a path in which the initial node coincides with the terminal node.

A tree is a graph which contains no circuits and has at most one branch entering each node. A root of a tree is a node which has no branches entering it, and a leaf is a node which has no branches leaving it. A root is said to lie on the first level of the tree, and a node which lies at the end of a path of length j-1 from a root is on the jth level. A tree which contains n roots is said to be n-tuply rooted, and if n = 1 it is called a rooted tree. The set of nodes which lie at the end of a path of length one from node x comprises the filial or sib set of node x, and x is the parent node of that set. The nodes within a sib set are sometimes referred to as sisters. The set of nodes reachable from node x is said to be governed by x and comprises the nodes of the subtree rooted at x. A chain is a tree which has at most one branch leaving each node. The number of branches leaving a node is called its branching ratio or degree. In particular, the degree of each leaf is zero.

A tree is said to be uniform if all nodes on level j are utilized (or "filled") before the tree is expanded to include nodes on level j + 1. Each level of the tree corresponds to one of the character sets S_i of a file. Such a tree is illustrated in Figure 6. Only the leaves will be of particular interest since they correspond to the items of a file. The remaining nodes are dummy nodes inserted to fill out the tree for search purposes.



The File $F = S_1 \times S_2 \times S_3$ where $S_1 = \{a,b,c\}$, $S_2 = \{d,e,f\}$, and $S_3 = \{g,h,i\}$ allocated in a uniform tree with $m = d = 3$ and $h = 3$. The sib set of node b is $\{bd,be,bf\}$.

Fig. 6

That set of nodes on level j which are the leaves of the subtree subtended by a node s on level j-k is called the kth sib set of node s and s is called the kth parent of the sib set. In general, the interest will be in first sib sets, which are those nodes "directly reachable" from a given node (the parent node), and, for brevity, first sib sets will be called just sib sets when no ambiguity will result. The members of a first sib set are called sibs. In a uniform d-way tree, (first) sib sets have exactly d members. The definition of sib set is extended so that the set of roots constitutes a sib set but the alternate term filial set is probably better in this case. In any listing of the members of a sib set, some node must be mentioned first.[1] That node will be called the heir node of the sib set. Sib sets are important because tree allocations are frequently characterized by the manner in which the sib sets are allocated.

In order to show the necessary memory space for storage, consider a tree of height h (the number of character sets) and n leaves (the number of items in the file). The common degree d (the average number of characters in each set) is such that $d^h = n$. In such a tree, the kth level has d^k nodes. Hence, the total number of nodes in a tree of height h is (See Appendix)

$$\sum_{k=1}^{h} d^k = d\,\frac{d^h - 1}{d - 1}$$

[1]No additional ordering relation on the character sets is assumed or implied.

If one word is used to represent one node and if n is assumed large,

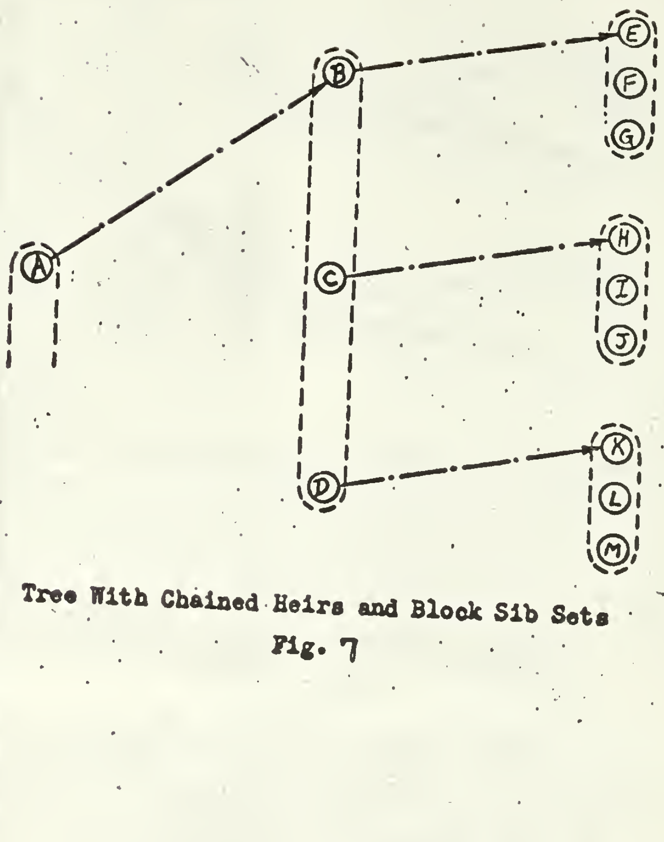
$$r(\text{words}) = \frac{d}{d - 1}$$

for the tree allocation system. However, there is no general one-to-one correspondence between nodes and words in computers with fixed word length.

The nodes of a tree may be grouped in many ways. One of the simpler ways is to chain each node to its heir (if it has one). The sib sets of which the heirs are members may then be stored in consecutive memory locations (blocks) in a random, ordered, or computable manner (Fig. 7). In addition, the sib sets themselves may also be chained together, so that the tree has two types of chains associated with it: one joining heirs and one joining sib sets (Fig. 8). A third method is to store heirs in memory blocks and to chain the sib sets (Fig. 9). One other useful chain for file-processing is a chain with similar characters of the same character set (tree level). See Figure 10. The three types of tree chains are called heir chains, sib chains and character chains respectively.

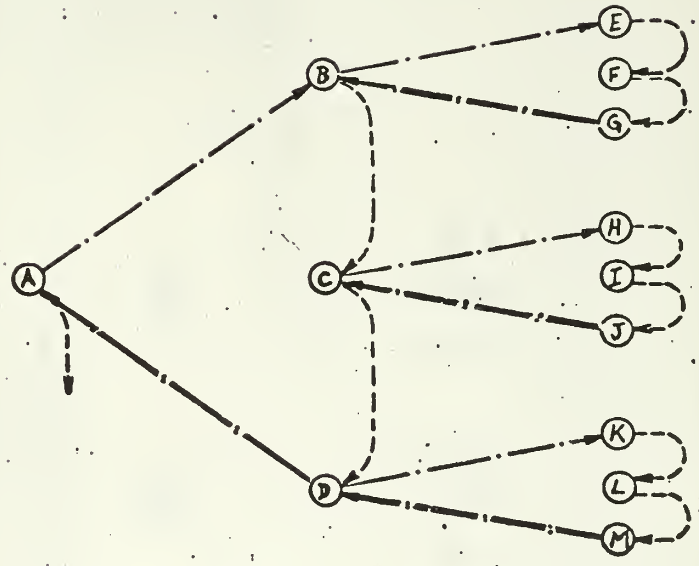
The character chains are superfluous for tree structures and, hence, need be included only to speed up various searches.[1] However, the concepts of heir and sib sets are essential to the tree structure. If heirs are grouped into consecutive memory locations and the sib sets chained, the resulting structure strongly resembles Iverson's

[1]See page 75 for searches with the mask vector $\underline{u}$ not a full vector or a prefix vector.



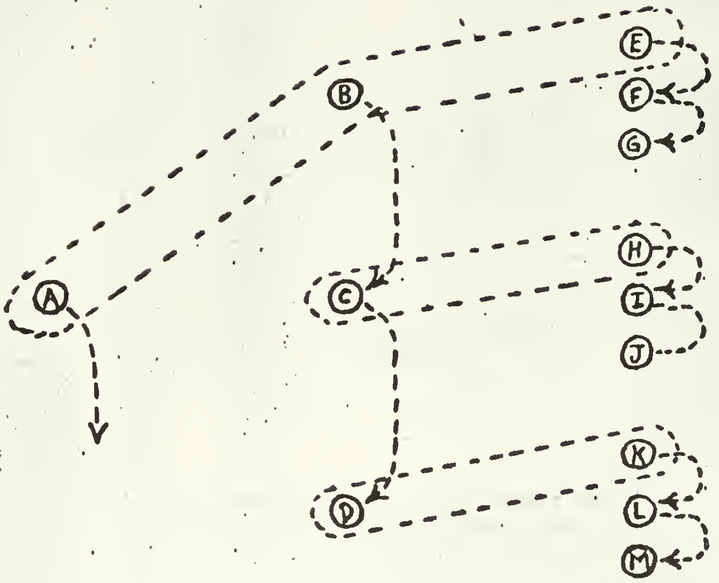
Tree With Chained Heirs and Block Sib Sets

Fig. 7



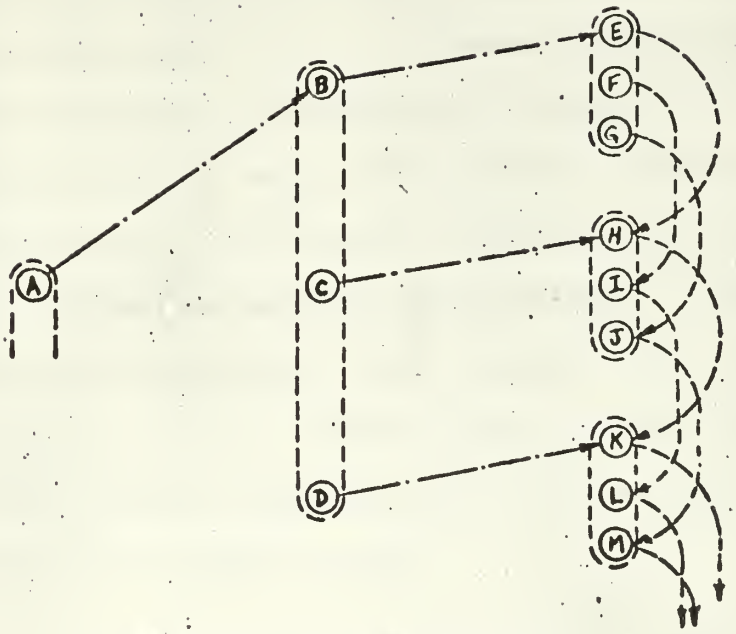
Tree With Chained Heirs and Chained Sib Sets

Fig. 8



Tree with block heirs and chained sib sets

Fig. 9



Tree With Chained Heirs, Block Sib Sets, and Chained Characters

Fig. 10

right list. [16] Indeed, it is exactly a right list if the heir blocks are stored in the proper order, i.e., if the chain is determined by the right list degree vector. Similarly, if sib sets are grouped into blocks and the heirs chained, the structure resembles a left list.

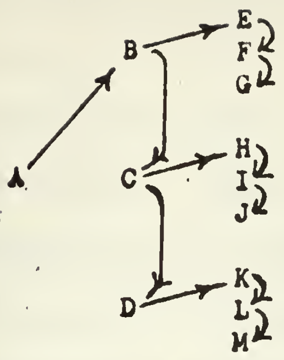
Each leaf of the tree corresponds to an item of the file. The intermediate nodes are dummies inserted to fill out the tree and form a path which is traced from root to leaf via the heir and sib links. The nodes on level i are identified with set S_i, and within each sib set on level i there are enough nodes to correspond to the characters of set S_i. There are at most m in each sib set, and there are d on the average. The memory word or words corresponding to a given node contains the representation of the character associated with the node (plus any chaining addresses and data that may be needed). The _label_ or _name_ of a node is the concatenation of the characters of the path leading to that node from a root.[1] Thus the label of a leaf is just an item (In Fig. 5 the character associated with a node is underlined and the remaining part of its label is not).

[1]It should be noted at this point that the labels of nodes are not always simple characters or concatenations of characters (alphabetic or numeric). In some applications of tree structures, the labels are entire words or strings of words. The problem of word length is more acute in these cases but the principles of manipulation are the same. Therefore the tree discussion will, for clarity, be in terms of simple characters.

A. Methods for computer storage.

For the purpose of computer storage of a tree structure, it is useful to consider the label or name of each node and its associated location-relation parameters as a vector. This vector is then considered as the jth row of a chain list matrix. See Figure 11. The nodes, and thus the rows of the matrix, may be listed in three different orders: 1) arbitrary (random) order, for chained heirs and chained sib set type trees, 2) level order, for chained heirs and block sib set type trees, and 3) subtree order for block heirs and chained sib set type trees. See Figure 11. The matrix representations of tree structures in Figure 11 include additional relationships which serve to simplify programming. Indeed, when the nodes are listed in either level or subtree order, the second and the third columns (names and degrees) of the matrices L and S may be shown to be sufficient. [32]

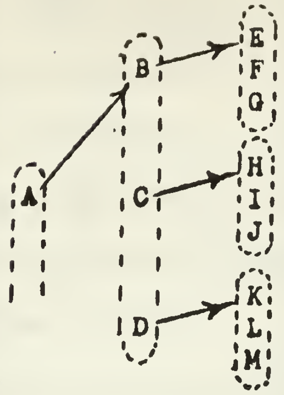
The big advantage, of course, to the arbitrary or random ordered matrix is that it is easy to add or delete nodes and effect other structural changes as required in many dynamic systems. No reshuffling is needed at any time. If the nodes are listed in level or subtree order, much of the structure is inherent in the order of listing and less memory is wasted on structure information. However, a severe price is sometimes paid when information too lengthy for the available extra space is inserted thereby requiring relocation of major portions of the matrix.



1	D	5	7	-
2	E	3	-	8
3	B	5	2	9
4	H	9	-	10
5	A	-	3	-
6	M	1	-	-
7	K	1	-	12
8	F	3	-	11
9	C	5	4	1
10	I	9	-	13
11	G	3	-	-
12	L	1	-	6
13	J	9	-	-

Chain list matrix A in arbitrary order. Chained heirs and chained sib sets.

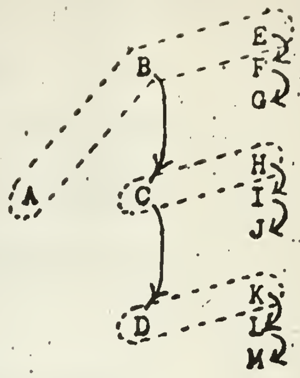
A_1	index
A_2	name (or label)
A_3	root on previous level (parent node)
A_4	heir chain
A_5	sib set chain (sister nodes)



1	A	3	6
2	H	0	-
3	I	0	-
4	J	0	-
5	-	-	-
6	B	3	14
7	C	3	2
8	D	3	10
9	-	-	-
10	K	0	-
11	L	0	-
12	M	0	-
13	-	-	-
14	E	0	-
15	F	0	-
16	G	0	-

Chain list matrix L in level order. Chained heirs and block sib sets.*

L_1	index
L_2	name (or label)
L_3	degree
L_4	heir chain



1	A	3	-
2	B	3	6
3	E	0	5
4	-	-	-
5	F	0	11
6	C	3	12
7	H	0	9
8	-	-	-
9	I	0	10
10	J	0	-
11	G	0	-
12	D	3	13
13	K	0	15
14	-	-	-
15	L	0	16
16	M	0	-

Chain list matrix S in subtree order. Block heirs and chained sib sets.*

S_1	index
S_2	name (or label)
S_3	degree
S_4	sib set chain (sister nodes)

Figure 11

* One space is left vacant at the end of each block. In general the number of of vacancies is <u>not</u> one. Rather it is a decision on the part of the programmer whether to leave space for expansion or not, and how much. The decision is based on the characteristics of the file and its intended use, e.g. size, frequency of search vs. frequency of addition and deletion, etc.

The factor of computer word-length enters into the consideration at this point. As we have previously pointed out, computers that have variable word lengths free the programmer from any concern over the row vector fit. When dealing with fixed word length machines, inefficiencies will be inevitable when the row vector length is not an even multiple of the computer word length.

Memory access systems for processing variable length operands in fixed word length computers have been proposed as a solution to this problem. Henderson and Flynn have suggested a scheme which addresses the characters of the memory words vice the words themselves. [7] A control triple contains the character address, the number of characters or length of the operand, and a single bit which signals right or left justification. Masking and shifting operations are performed in the hardware and entail no time penalty.

As an example of the necessary word length computation, consider again the row vectors of the matrices in Figure 11. The indices in A_1, L_1, and S_1 are the addresses of the computer words containing each row vector. If we allow the names of nodes to be any alphanumeric, then b = 6 bits and $m \leq 2^b = 64 \geq d$. We pointed out previously (p.60) that in some applications of trees the names of the nodes are more complex than a single alphanumeric character. In these cases the character may be replaced by the address of a list which may be any length ≥ 1. See Case 2 below:

Case 1: Alphanumeric name

A_2	6	L_2	6	S_2	6
A_3	115	L_3	6	S_3	6
A_4	15	L_4	15	S_4	15
A_5	15		27 bits		27 bits
	51 bits				

Case 2: Complex name

A_2	15	L_2	15	S_2	15
A_3	15	L_3	6	S_3	6
A_4	15	L_4	15	S_4	15
A_5	15		36 bits		36 bits
	60 bits				

Two distinct cases occur when dealing with multi-tree structures. The first is the case in which all the nodes are considered distinct. The second is the case in which many nodes exist in more than one rooted subtree. Note that the nodes <u>within</u> each rooted subtree are considered distinct. Due to the greater complexity of the minimum matrix representation for the second case, its use should be considered only when the number of nodes which are unique to one given tree is less than one half the total number of nodes. [19]

A heirarchical tree is an example for which it is expected that each node will appear only once, while syntactical trees representing sentence structures is an example for which a large number of nodes (words) will be repeated.

The problem of non-distinct or overlapping nodes is basic in the application of list and tree structures. One of the important merits of list processors is that data having multiple occurrences often need not be stored more than one place in the computer. One may visualize this situation as the overlapping or intersection of

lists, and its utilization may be regarded as a step toward global optimization of storage allocation.

However, overlapping lists pose a problem when it becomes necessary to erase a particular list or node and return the space to the list of available space (LAS). When a given list is no longer needed, it is desired to erase just those parts that do not overlap other lists still in use. There is no general method of doing this short of making a survey of all lists in memory. [3]

Several solutions to this problem have been found, each with its own disadvantages or limitations. A solution incorporated by McCarthy in his LISP merely abandons lists as they are no longer needed. [22] Then, when the LAS has been depleted, a survey is made of all registers currently employed in non-abandoned lists and the complement of this set is returned to the LAS.

There are two sources of inefficiency in this method. Depending on the computer, the application, the programmer's skill, etc., the time needed to reclaim unused storage space is nearly independent of the amount of space reclaimed. The efficiency, therefore, drops off rapidly as the memory approaches capacity. Second, the method as used by McCarthy required that a bit (in this case, the sign bit) be reserved in each word for tagging accessible registers during the survey of accessibility. If data consists of signed integers or floating-point numbers, this results in awkwardness and further loss of efficiency. This method can be modified by setting aside a block of memory and establishing a one-to-one correspondence between the bits in this block and the words in the remaining memory. This

[illegible]

modification eliminates the awkwardness of arithmetic, but takes a further toll in the speed of reclamation. [3]

A second method, used by Gelernter et al in FLPL, consists of adopting the convention that each list component is "owned" by exactly one list, and "borrowed" by all other lists in which it appears. [10] A "priority bit" is then used in each "location word" to indicate whether the list entry referenced is owned or borrowed. An erasing routine is used which erases only those parts of a list that are owned. This system obviously works only so long as no list is erased that owns a part that has been borrowed by some other list not yet erased.

A third method has been developed by Collins as a means to overcoming the disadvantages of the first two. [3] The method is based on the interspersion of words containing reference counts. Viewed in terms of the conventional diagrams of lists, such a reference count is a tally of the number of arrows point <u>to</u> the box containing the reference count and emanating <u>from</u> boxes in the same or other lists. See pp. 5[illegible], 6[illegible]. A two-bit type code then appears in each location word[1], one value of which serves to identify

[1]Each list is represented by a set of location words and data words. Data words contain atoms and consist of a single field. Location words are divided into fields which contain a <u>type code</u>, the address of the next location word, and a field which may contain any one of four things, determined as follows by the type code:

0	Location of an atom
1	Atom (if max. no. of bits small enough)
2	Location of a list
3	Reference count

a reference count as such. The obvious convention is adopted that, unless a word contains a reference count, there is exactly one arrow pointing to it. A reference count of one is thereby redundant but otherwise harmless.

A disadvantage of this system is that atoms cannot be borrowed. Often this will not be a significant encumbrance. When it is, it can be partially solved by uniformly replacing each atom, a, by a list whose only term is a.

Finally, a method slightly different from the solutions already given is the use of an auxiliary aid to keep tabs on distinct nodes and successor pairs. This aid is called a <u>connection matrix</u>. A clear description of this matrix is too lengthy to be given here. Suffice to say that it exists apart from the chain list matrix which describes the tree but the chain list matrix in this case contains only distinct nodes. See [32] for a brief description and [19] for a more rigorous treatment.

B. Search operations for trees.

We previously discussed search operations when data was stored according to any one of the four basic schemes, i.e. random, ordered, computable and chained allocations. The same searches may be applied to the tree allocations with minor modifications since the tree allocations employ various combinations of the basic allocations. We shall again consider:

Match search

Multiple-match search

Smallest search

Next smaller search

Add an item

Delete an item

It will be assumed that the tree under consideration is uniform for the derivation of efficiencies but this restriction does not apply to the search procedures themselves. As in Section 4, the treatment here is adapted from Sussenguth with minor modifications. [36]

To conduct a match search in a tree allocation, one first searches the sib set of roots to locate the node whose associated character is the same as the first character of the argument. Then one proceeds to the first sib set of this node and searches again to locate the node with character matching the second character of the argument. This process continues through level h. The manner in which the sib sets are searched and the manner in which the next sib set is located depend upon the particular allocation.

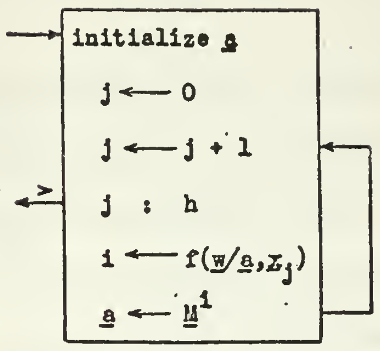
Consider first the allocation in which heirs are chained and the sib sets are computable allocations within blocks. For a match search, h words must be retrieved from memory, one for each level. The address of the root word is computed for the first element of the argument and an initial address parameter. The root word will contain the parameter required to locate its sib set, and this parameter and the second element of the argument are used to compute the address of the word corresponding to the proper node

on the second tree level. This process is repeated for all h levels (program T1), thereby entailing h memory accesses and h function evaluations.

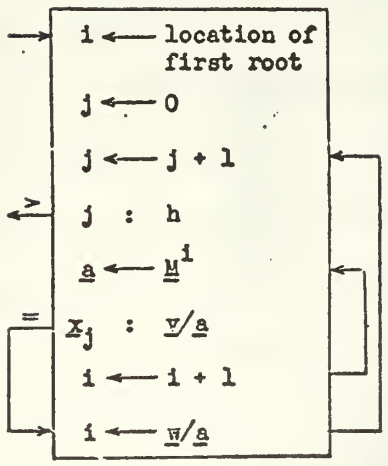
As a simple example of such an allocation consider the character sets to be the integers from 0 to m-1, and let the sib sets be stored in order. At each node is stored the address α of its heir node. Thus, if the node of the kth level has just been retrieved, the address of the node of the (k+1)th level is just $\alpha + \underline{x}_{k+1}$ where $\underline{x}$ is the argument of the search.

An allocation similar to this is that in which the heirs are chained but the sib sets are random allocations within blocks. The chaining from node to heir is accomplished by extracting the chain address which is stored at each node. The proper node of the sib set is obtained by comparing the stored character with the appropriate character of the argument; if they match, the proper node of the sib set has been found; if they do not match, the next memory word, which is the next node in the sib set, is tested. When the proper node has been found, its heir is located. This is repeated for all h levels (program T2). Thus, the search time is composed of the h item times needed to pass from root to leaf via the heir chain plus the item times needed in the sib set searches. These are at most d-1 per set, at least none per set, and (d-1)/2 per set on the average.

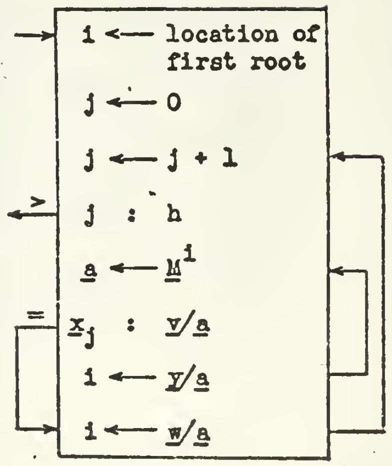
The search procedure for the allocation in which both the heirs and the sib sets are chained is exactly the same as the search just described except that, when searching the sib sets, the sib chaining



Program T1
Match search for $\underline{x}$
Heir chains, computable sib sets



Program T2
Match search for $\underline{x}$
Heir chains, random sib sets



Program T3
Match search for $\underline{x}$
Heir chains, sib chains

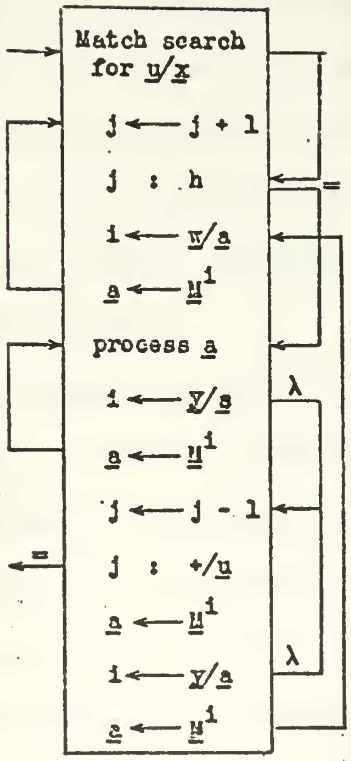
address is used to locate the next node rather than consecutive memory locations. (Notice that program T3 is program T2 but for one line).

Because these three allocations are so similar, the details of the remaining searches will be discussed only for doubly-chained trees.

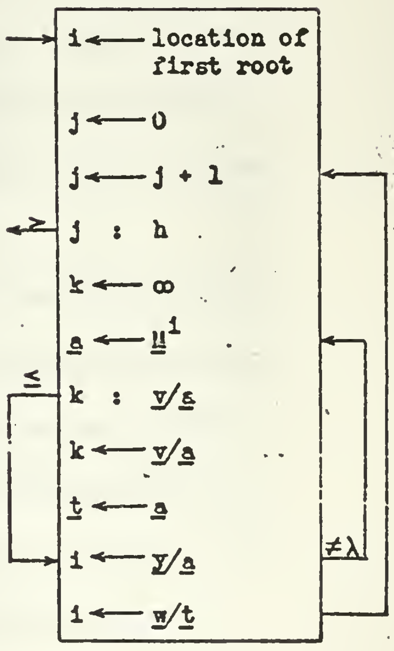
The items satisfying the multiple-match search when $\underline{u}$ is a prefix vector are those items corresponding to the subtree subtended by the node with label $\underline{u}/\underline{x}$. Program T4 delineates such a search: First, the node $\underline{u}/\underline{x}$ is located by a match search. The leaves of its subtree are found by finding the heir node of $\underline{u}/\underline{x}$, then the heir of the heir, etc., until the heir which is a leaf is found; its sibs are retrieved. Then, it is necessary to "back-up" one level to the parent node of the sib set just retrieved, find the next node of the sib set of that level, move "forward" to its heir, and retrieve its associated sib set. This process is repeated until the entire subtree has been retrieved. Thus, if the $\underline{u}/\underline{x}$ node is in level h-k (i.e. $\underline{u}$ has h-k 1's) the d^k subtended leaves are retrieved by locating d^{k-1} sib sets; the parents of these sib sets, in turn, form d^{k-2} sib sets themselves, and so on back to the $\underline{u}/\underline{x}$ node. Thus to retrieve the $s = d^k$ subtended leaves,

$$2\sum d^i + d^k = \frac{d^{k+1} + d^k - 2d}{d-1} = \frac{s(d+1)}{d-1}$$

items must be handled. (The factor of 2 occurs because each non-leaf node is retrieved twice, one in the forward direction and once



Program T4
Multiple-match search
for $\underline{x}$ wrt prefix $\underline{u}$
Heir chains, sib chains



Program T5
Smallest search
Heir chains, sib chains

in the backup.) The match search time to retrieve the $\underline{u}/\underline{x}$ node must be added to this to determine the multiple-match search time.

To accomplish this search, it is necessary to locate the parent of a given node; the chaining structure introduced thus far does not permit this. However, a simple modification can provide this facility: The last node in each sib set has nothing chained to it in the sib chain; therefore, its address portion is vacant and may be used to indicate the location of the parent of the sib set. To avoid an ambiguity[1] of having a "backward-heir" chain address in the portion of the word assigned to sib chains, it is necessary to add an extra marker bit to the sib chain addresses which indicated to which chain the address belongs. These "backward-heir" or parent chains are indicated by two broken lines in Fig. 8 and the marker bit is denoted by the symbol λ in the programs.

The tree allocations with no sib chains cannot have the back-up property as described. Frequently, however, this may be circumvented by additional bookkeeping in the search programs, which may entail a considerable expenditure of time with respect to the retrieval of a single address. If there is sufficient word length available, an additional link may be included to indicate the parent node as illustrated in Figure 11, Column A_3.

[1]There is no ambiguity if the number of nodes in the sib set is known. This will not, however, be the case in general.

The smallest search in the doubly-chained tree allocation proceeds exactly as the match search, except that the smallest character in each sib set is selected rather than a matching character. (program T5).

To search for the item next smaller than a given item is more complex, however. One first locates the given item; then its sibs are searched to find a smaller item. (Specifically, since the words contain only a single character, the sib set is searched for the character next less than the least significant character of the argument). If such an item exists, the search is completed. If not, that is if the given item is the smallest in the set, the parent node must be retrieved and its sibs searched for a smaller character. If one is found, its first sib set is searched for its largest item and the search ends. However, it may still be necessary to back-up another level.[1] Thus, if k back-ups are required, the search is composed of one match search, 2k links in heir chains (k in each direction), and next smaller or largest searches through 2k + 1 sib sets (the original sib set of leaves, k searched for the next smaller item, and k for the largest).

Since about d item times are required to search a sib set, the search times may be calculated once the appropriate values for k are known. Clearly, the minimum k is 0, and the maximum h - 1 (the argument is the smallest item of the file). To ascertain the

[1]In the chained allocation at most one back-up was needed because the entire item could be examined and the most significant character changed. Here several back-ups may be needed because only one character may be examined at a time.

average k, consider the character sets to be the integers between 0 and d-1. In the file there are $d^{h-k} - d^{h-k-1}$ items with exactly k trailing 0's, and for each such item k back-ups are necessary. Thus[1]

$$k_{avg} = \frac{1}{d^h}\sum_{i=0}^{h-1}(d^{h-i} - d^{h-i-1})i$$

$$= (1 - \frac{1}{d})\sum_{i=0}^{h-1}\frac{i}{d^i}$$

$$= \frac{1}{d^h}\left(\frac{d^h - 1}{d - 1} - h\right)$$

$$\doteq \frac{1}{d - 1}.$$

In heir-chained tree allocations the procedures for adding or deleting items are analogous to the procedures for computable, random, or chained files according as the sib sets are computable, random, or chained.

The tree allocation schemes as described thus far allow efficient searches only when the mask vector $\underline{u}$ is either the full vector or a prefix vector. For example, to perform a match when $\underline{u} = (0,1,\ldots)$, it is necessary to search all of the nodes of the second level to locate characters matching the second character of the argument. To test all the second level nodes, it is necessary to proceed from a root to its sib set, back to the root, down to the next root, to its sib set, etc. This could be a time-consuming process, especially

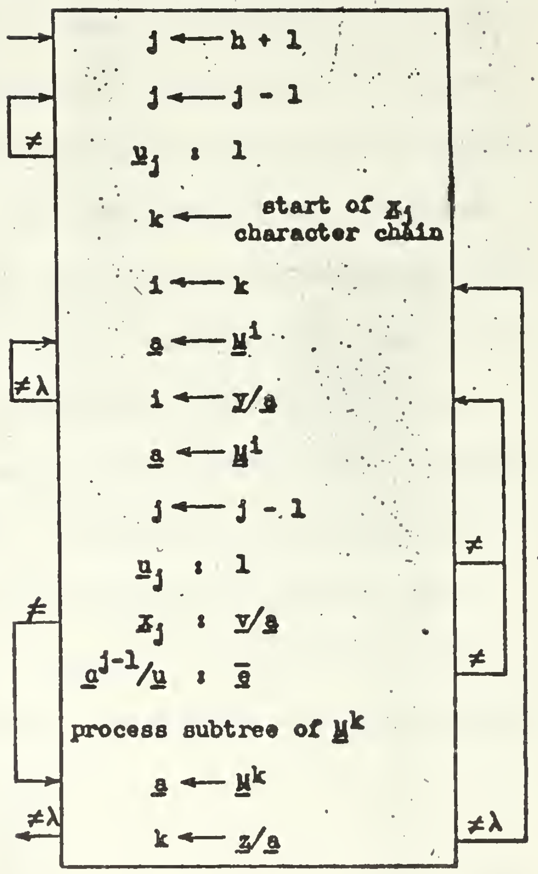
[1]See appendix.

if $\underline{u}$ had several leading 0's.

If, however, the allocation includes character chains (Figure 10), arbitrary $\underline{u}$ vectors are permissible. Consider the multiple-match search $\underline{x}$ wrt $\underline{u}$ for a tree which has heir, sib, and character chains. Assume $\underline{u}$ has 1's in positions $\underline{u}_{j1}, \underline{u}_{j2}, \ldots, \underline{u}_{jk}$ and 0's elsewhere. The character chain of $\underline{x}_{jk}$ is traced to its first node. Then the $(j_k - j_{k-1})$th parent of that node is found by proceeding via the parent chain for $j_k - j_{k-1}$ levels. If that parent node does not have character value $\underline{x}_{jk-1}$, this portion of the search may be terminated, and the next node of the character chain tested. If the parent node does have value $\underline{x}_{jk-1}$, however, find its parent in the j_{k-2}th level and test it similarly. If the original candidate node is such that all its parents in levels $j_1, j_2, \ldots, j_k$ have character values matching $\underline{x}$, then the leaves of the subtree subtended by that node constitute part of the set of items satisfying the search (program T6).

C. Evaluation of results.

Tree allocation systems appear both efficient and versatile. Most files can be allocated in a tree structure without difficulty. One set, say S_1, is chosen to correspond to the roots, and all items having the same first character are placed in the subtree subtended by the root corresponding to that character. Sib sets in the second tree level are formed from the characters of another set, say S_2, and then their sib sets, through h levels. If the file items are correlated (i.e. have common characters), and it is reasonable to expect that such correlation would exist in practical applications, the number of dummy nodes which must be added is relatively



Program T6
Multiple-match search
for $\underline{x}$ wrt arbitrary $\underline{u}$

small.[1] Indeed, for many trees fewer bits are required to store the file and its associated structure than are required to store the file on an item-by-item basis. This apparent paradox vanishes when it is observed that one character at the root of the tree is shared by all the items of the subtree subtended by that root. Moreover, in a tree allocation it is possible to perform all of the operations considered with relative ease. More complex operations (e.g. search for the item next smaller than a given item with respect to an arbitrary permutation) can be accomplished with more elaborate programs.

When comparing the relative efficiencies of the tree allocations the same considerations regarding like quantities as were discussed before apply (p.43) , i. e. time and required storage capacity with due regard for their inverse relationship. Table 4 is a summary of search times and storage ratios. As in Table 2, there are three entries for each search: the maximum, minimum, and expected number of item times to complete the search. Frequently the entries are approximations to the expressions which have been derived. When applicable, the $\underline{u}$ and $\underline{p}$ for which the entries are calculated are shown. Table 5 repeats the average values of Table 4 but with typical values chosen for the file parameters.

[1] It is easy to show this number, which is $n(1-r(\text{words}))$, is minimized if the set of smallest order is chosen to correspond to the roots, the set of next smallest order to the second level, etc.

[illegible]

		Tree (heir chains, computable sibs)	Tree (heir chains, random sibs)	Tree (heir chains, sib chains)
Max	Search for Match	$h(f + 1)$	hd	hd
Min		$h(f + 1)$	h	h
Avg		$h(f + 1)$	$\frac{h}{2}(d + 1)$	$\frac{h}{2}(d + 1)$
$\underline{u}$	Search for Multiple Match	(1)	(1)	(1)
Max		$h + f(h - k + \frac{s - d}{d - 1})$	greater than the corresponding times of tree with heir and sib chains	$hd + s\frac{d + 1}{d - 1}$
Min		$h + f(h - k + \frac{s - d}{d - 1})$		$h + s\frac{d + 1}{d - 1}$
Avg		$h + f(h - k + \frac{s - d}{d - 1})$		$(d + 1)(\frac{h}{2} + \frac{s}{d - 1})$
$\underline{u}$ $\underline{p}$	Search for Smallest	(1) (2)	(1) (2)	(1) (2)
Max		$h(f + 1)$	hd	hd
Min		$h(f + 1)$	hd	hd
Avg		$h(f + 1)$	hd	hd
$\underline{u}$ $\underline{p}$	Search for Next Smaller	(1) (2)	(1) (2)	(1) (2)
Max		$h(f + 1)$	greater than the corresponding times of tree with heir and sib chains	$3hd + 2h - d - 2$
Min		$h(f + 1)$		$h + d$
Avg		$h(f + 1)$		$(d + 1)(\frac{h}{2} + \frac{2}{d - 1}) + d$

Summary of Search Times and Storage Ratios

Table 4

(Continued on next page)

		Tree (heir chains, computable sibs)	Tree (heir chains, random sibs)	Tree (heir chains, sib chains
Max	Add an Item	large	$(h + 1)d$	$hd + 1$
Min		large	$h + 1$	$h + 1$
Avg		large	$\frac{1}{2}(hd + h + d)$	$\frac{1}{2}(hd + h + 2)$
Max	Delete an Item	$h(f + 1)$	$(h + 1)d$	$hd + 1$
Min		large	$h + 1$	$h + 1$
Avg			$\frac{1}{2}(hd + h + d)$	$\frac{1}{2}(hd + h + 2)$
Storage Ratio (words)		$\frac{d}{d - 1}$	$\frac{d}{d - 1}$	$\frac{d}{d - 1}$
Storage Ratio (bits)		$\frac{a}{bh} \frac{d}{d - 1}$	$\frac{a + b}{bh} \frac{d}{d - 1}$	$\frac{2a + b + 1}{bh} \frac{d}{d - 1}$
Bits per Word		a	$a + b$	$2a + b + 1$

Mask vector $\underline{u}$ restriction

1. mask must be prefix of tree (level) order

Permutation vector $\underline{p}$ restriction

2. identity permutation only

Table 4 (continued)

	Tree computable sibs	Tree random sibs	Tree chained sibs
Match	18	19	19
Multiple match	54	> 166	166
Smallest	18	31	31
Next smaller	18	> 27	27
Add item	large	21	20
Delete item	large	21	20
Bits per word	15	21	37
Storage ratio (words)	1.24	1.24	1.24
Storage ratio (bits)	.52	.72	1.27

Parameters		all characters implicit
a = 15	h = 6	
b = 6	m = 30	
c = 3	n = 20000	
f = 2	s = 100	

Comparative item times for the typical parameters shown.

Table 5

7. Minimization of expected search time, and the search-time, storage-space product.

We have seen the general reduction in item times with only a slight increase in storage ratios when considering tree allocations vs. the four basic allocations. Note Tables 2 and 3 vs. Tables 4 and 5. The item times given in Table 5 are dependent on the selection of the typical parameters shown just beneath the table. It is the purpose of the following discussion to show some calculations based on these parameters which will optimize the desired characteristics of the tree structure.

The parameters shown in Table 5 are defined in Table 1 and are reiterated here for convenience:

$a = 15$	number of bits to specify an <u>a</u>ddress
$b = 6$	number of <u>b</u>its to specify a character
$c = 3$	number of character positions which are <u>c</u>hained
$f = 2$	number of item times to compute <u>f</u>unction f
$h = 6$	number of character sets (<u>h</u>eight of the tree)
$m = 30$	number of <u>m</u>embers of each character set
$n = 20,000$	number of items in the file
$s = 100$	number of items satisfying a multiple-match <u>s</u>earch

Parameters a and b are characteristics due to binary coding and the machine in use. c and f are included to allow calculations for the "Tree with computable sibs" column. We disregard these now since the difficulty with <u>add item</u> and <u>delete item</u> functions make this scheme unfeasible except in very limited applications. n and s are characteristics of the file used for comparison and are therefore beyond our control. This leaves h and m, the only two

parameters available for manipulation.

If there are m nodes in a filial set, the expected number which must be tested before a match between a node value and the appropriate element of the query is ½m. Thus the expected number of chaining links required to find a match within one filial set is ½ (m+1); one link to reach the filial set and ½ (m - 1) to search it.

Assuming the search time is proportional to the number of chaining links traversed, the expected search time may be calculated if all the filial set sizes are known. However, it is inconvenient to use the set of all filial set sizes in macroscopic calculation. For computational purposes an average filial set size for each tree level is defined as:

$$m_i = \frac{\text{number of nodes on level } i}{\text{number of filial sets on level } i} \qquad (1)$$

Since the average time to search a filial set on the ith level is $\frac{1}{2}(m_i + 1)$, the expected time to search a file of n items allocated as an h level tree is

$$t = \frac{1}{2} \sum_{i=1}^{h} (m_i + 1). \qquad (2)$$

Equation (2) gives the expected search time in terms of the parameters m_i and h. These parameters are related by the expression

$$\prod_{i=1}^{h} m_i = n, \qquad (3)$$

because the product of the filial set sizes is the number of leaves.

[37]

The binary coding of alphanumeric characters allows additional flexibility in the variation of m_i provided the key elements may be manipulated freely. The key may then be considered to consist of a single string of binary digits rather than several disjoint elements, each consisting of several binary digits. For example, the key "CAT" is considered as the binary string "010011010001110011" rather than the set of distinct elements "010011", "010001", and "110011". With keys of this format, the binary digits may be grouped to give the most efficient search system, i.e. m_i and h may be selected without constraint (other than (3)) to minimize t. An example of this division technique is shown for several alphabetic characters in Figure 12. [37]

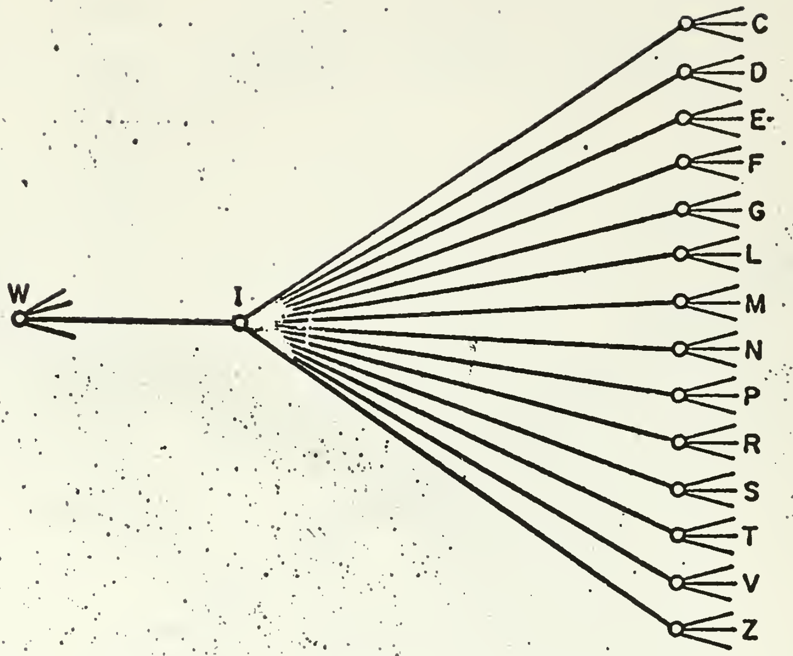
Noting the form of equations (2) and (3), it is clear that the expected search time is independent of the order in which the levels are taken and that the minimum search time is achieved when the m_i are all equal. If the m_i are all equal, (3) reduces to $m^h = n$ and (2) becomes

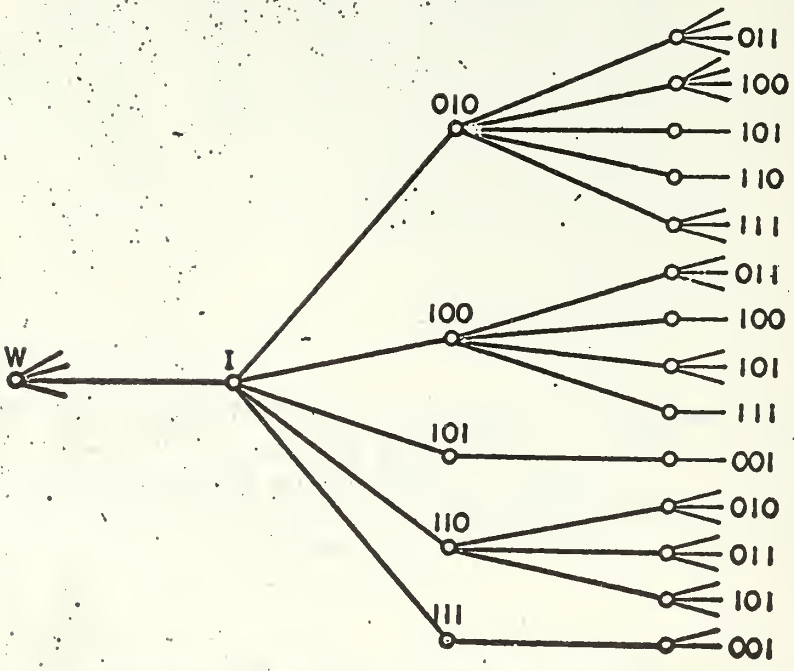
$$t = \tfrac{1}{2}\, h(m + 1) = \tfrac{1}{2}(m + 1) \log_m n \tag{4}$$

Taking the first derivative of this expression with respect to m and setting it equal to zero, we find a minimum t at $m = 3.6$. Equation (4) is normalized and plotted in Figure 13. By simple arithmetic we find

$$t = 2.3 \log_{3.6} n = 1.24 \log_2 n$$

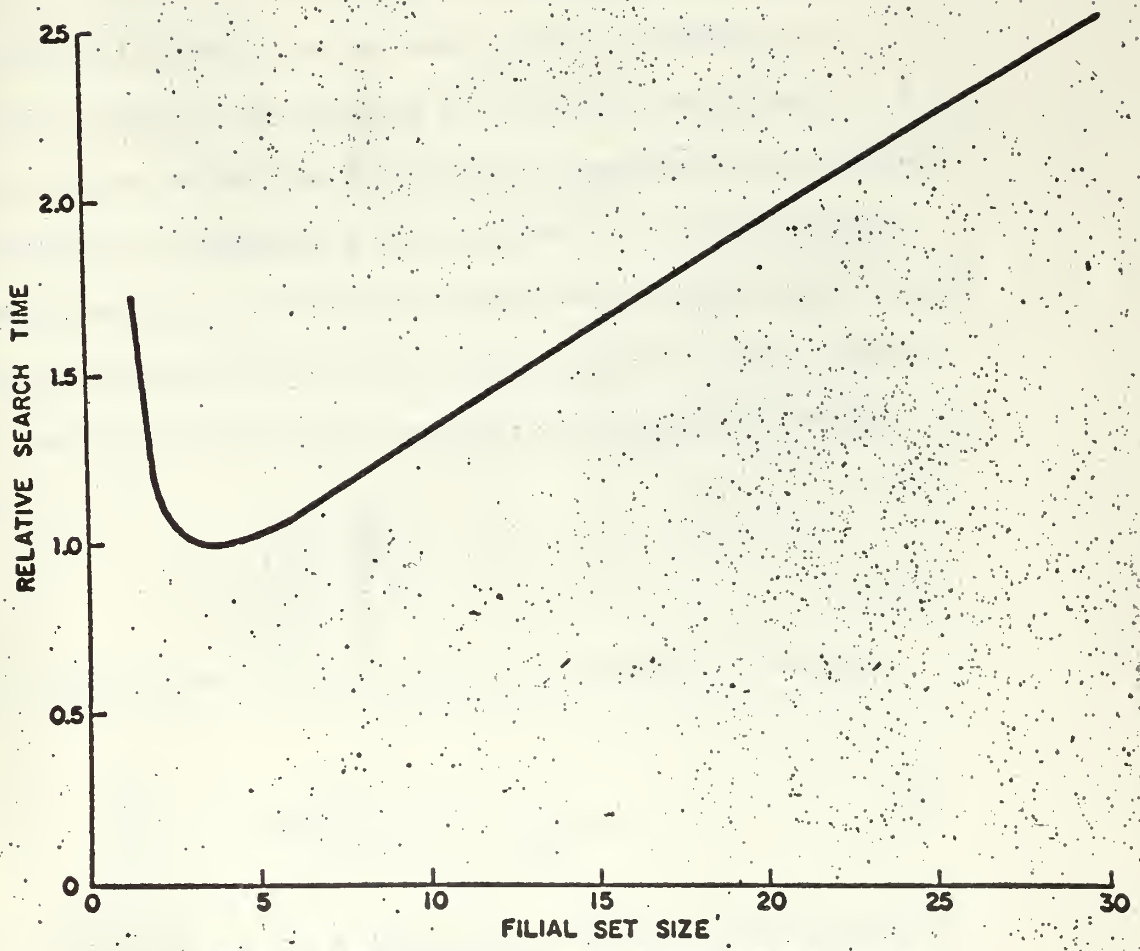
That is, the expected search time in the optimum case is only 24 percent slower than the binary search time. Moreover, since the





The Third Level of the Upper Tree is Split into Two Levels of the Lower Tree

Figure 12



Relative Search Time as a Function of Average Filial Set Size

Figure 13

time required to add or delete an item to or from the file is approximately the same as the search time, a considerable improvement over the file allocated for binary searching (ordered allocation).

Therefore in summarizing the case when it is possible to freely manipulate keys, the key elements should be selected so that

1) all paths from a root to a leaf have the same length, h;

2) all filial sets have the same number of members, m;

and 3) the common filial set size m is near 3.6 and $h = \log_m n$. [37]

The above rules for the adjustment of parameters are based on the consideration of minimum search time only. A more realistic value may be obtained by also considering the cost due to the higher storage ratio caused by small filial set size. The total number of (dummy) nodes in the tree which serve only to index the leaves (items) is

$$W = \sum_{j=1}^{h} \prod_{i=1}^{j} m_i \qquad (5)$$

If it is assumed that all filial sets m_i are equal (5) becomes

$$W = \sum_{j=1}^{h} m^j = m\left(\frac{m^h - 1}{m-1}\right) \cong \frac{m}{m-1}\, n \qquad (6)$$

Equation (6) shows that as m <u>increases</u>, the number of required storage locations <u>decreases</u>.

We have previously defined a meaningful measure of efficiency for storage allocations as the product of search time and storage capacity. Using (4) and (6)

$$C = t \times W = \left[\tfrac{1}{2} h(m + 1)\right] \times \left[m\left(\frac{m^h - 1}{m - 1}\right)\right]$$

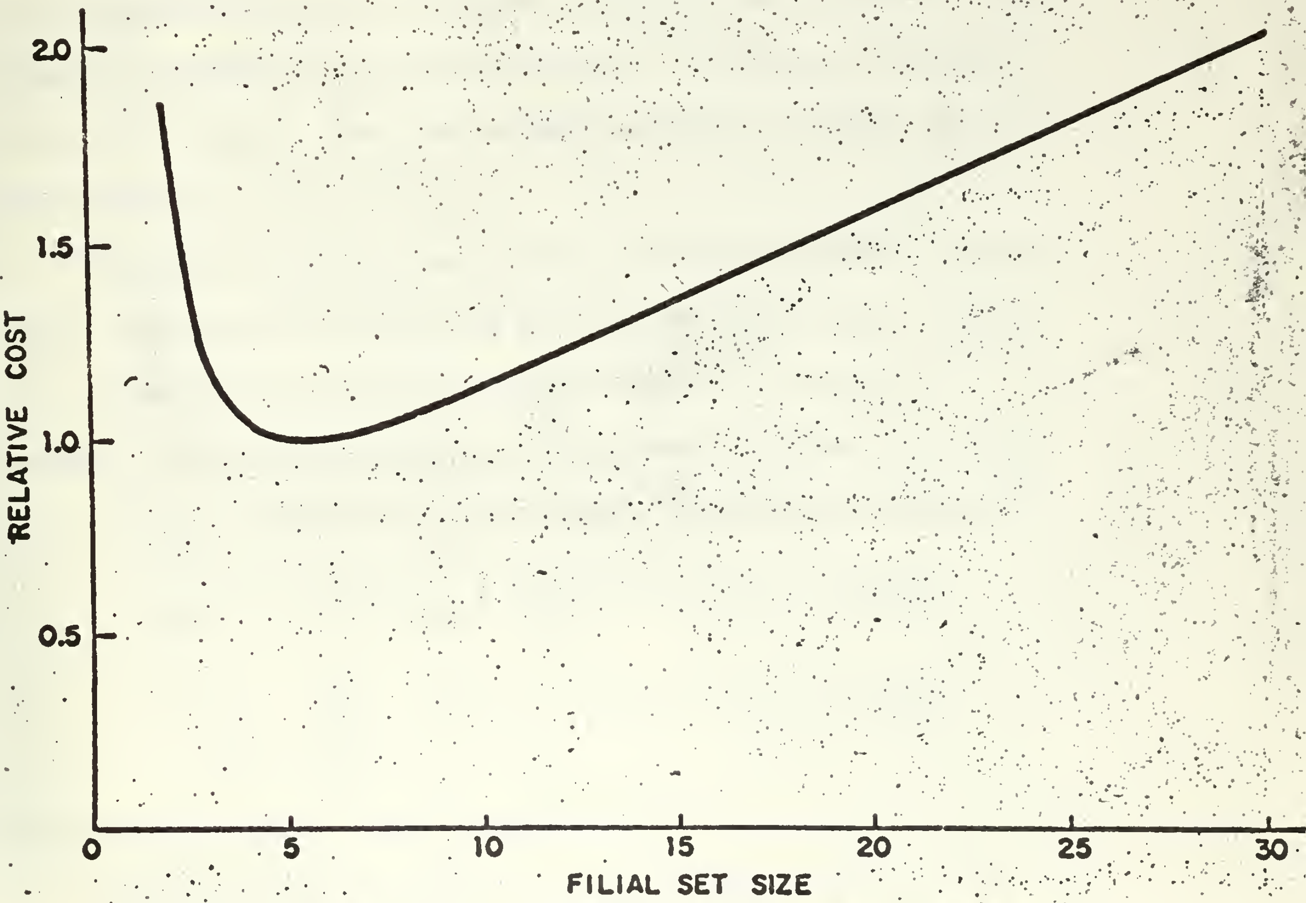
$$= \tfrac{1}{2} hm \left(\frac{m + 1}{m - 1}\right) (m^h - 1) \qquad (7)$$

Equation (7) is plotted in Figure 14 in normalized form. The minimum cost is achieved when m = 5.3.

It is useful to note that in both Figure 13 and Figure 14, the curves are shallow at the minima. The values of m may therefore vary from approximately three to eight with less than 20 percent degradation.

When it becomes necessary to use an m greater than these optimum values, the efficiency of the system will suffer. Numeric key elements would be acceptable since m = 10. But when the keys are alphabetic (m = 26) or alphanumeric ($m > 36$), it becomes necessary to add refinements.

In the discussion of efficiencies above, a tree with chained heirs and random chained sib sets was implied. We have seen that this sort of doubly-chained tree requires two address fields (one for the heir link and one for the sib set link) plus a field for the label. A slightly more complex allocation in which the sib sets are chained in an ordered arrangement (this will permit a binary type search within each sib set) will be efficient for larger m. To accomplish this ordering for a binary search within a filial set, it is necessary to expand the memory map (matrix row



Relative Cost as a Function of Average Filial Set Size

Figure 14

[illegible]

vectors) to three addresses per node, corresponding to the states: greater than, equal to, and less than. In the two unequal conditions the address is that of the next sister node to be examined; the equal condition address leads to the filial set entry at the next level, or if the termination symbol has been recognized, to the location of the desired item. [2]

On a variable word length computer this revision would demand an approximate 50 percent increase in storage requirements. Less flexible machines would probably require a 100 percent increase. Even in this latter case, the proposed method is superior as we shall now show.

Consider a filial set of size m. If the ith member of the set is selected as the starting point in the search, the original set is partitioned into three subsets with 1, m - n, and n - 1 members. The average search times for these sets are 1, 1 + T_{m-n} and 1 + T_{n-1} respectively. The average search time is therefore

$$T_m = \frac{1}{m} \left[1 + (1 + T_{m-n})(m-n) + (1 + T_{n-1})(n-1) \right]$$

$$= 1 + \frac{1}{m^2} \sum_{n=1}^{m} \left[(T_{m-n})(m-n) + (T_{n-1})(n-1) \right]$$

Because of symmetry, this is equivalent to

$$T_m = 1 + \frac{2}{m^2} \sum_{n=1}^{m} (T_{n-1})(n-1)$$

If we define i = n - 1, we have

$$T_m = 1 + \frac{2}{m^2} \sum_{i=0}^{m-1} i T_i$$

which is equivalent to

$$T_m = 1 + \frac{2}{m^2} \sum_{i=1}^{m-1} i T_i$$

Obviously, $T_1 = 1$.

Now that the expected search time is known, relative cost can be computed on the same basis as used previously (equation (7)) for direct comparison. See Table 6 for a comparison of average search time, relative search time, and relative cost vs. the size of the filial sets. With a 50 percent increase in storage requirements assumed in computing relative cost, the proposed method is superior when the average filial set size exceeds 9. If 100 percent storage increase is required, the break-even point is 16. [2]

Another variation is to allow unbalanced trees. This type of tree may be utilized when it is not possible to freely manipulate keys and select filial sets to be near optimum size. The path lengths may then vary within the tree, and the tree may be constructed using random length key words. In the case where the main storage is a disc, the proper path length is easily determined because each leaf must govern T items (where T is the number of items that may be accommodated on one track of the disc file). Thus, if a particular node governs T or fewer items, the branching along that particular path may be stopped. If, however, the node governs more than T items, the branching must be continued for at least one more level.

TABLE 6

Size of Filial Set	*Average Search Time*		*Relative Search Time*		*Relative Cost*	
	Old	*New*	*Old*	*New*	*Old*	*New*
2	1.5	1.500000	1.2052	1.2052	1.8590	2.7886
3	2.0	1.888888	1.0138	.9575	1.1729	1.6616
4	2.5	2.208333	1.0043	.8871	1.0328	1.3684
5	3.0	2.479999	1.0381	.8581	1.0008	1.2410
6	3.5	2.716666	1.0878	.8444	1.0068	1.1722
7	4.0	2.926530	1.1448	.8375	1.0301	1.1304
8	4.5	3.115177	1.2052	.8343	1.0623	1.1031
9	5.0	3.286595	1.2673	.8330	1.0996	1.0842
10	5.5	3.443729	1.3302	.8329	1.1400	1.0706
12	6.5	3.723622	1.4568	.8345	1.2257	1.0532
14	7.5	3.967632	1.5827	.8372	1.3146	1.0431
16	8.5	4.184048	1.7073	.8404	1.4046	1.0371
18	9.5	4.378560	1.8304	.8436	1.4948	1.0334
20	10.5	4.555252	1.9520	.8468	1.5847	1.0312
25	13.0	4.937190	2.2492	.8542	1.8070	1.0294
30	15.5	5.256304	2.5380	.8606	2.0250	1.0300
35	18.0	5.530518	2.8196	.8663	2.2386	1.0317
40	20.5	5.771009	3.0949	.8712	2.4482	1.0338
45	23.0	5.985222	3.3649	.8756	2.6542	1.0360
50	25.5	6.178372	3.6302	.8795	2.8570	1.0383
55	28.0	6.354258	3.8913	.8830	3.0568	1.0405
60	30.5	6.515730	4.1487	.8862	3.2540	1.0427

Frequently there are subtrees which are also chains, i. e. the filial set of each node in the subtree consists of a single node. For example, when using English language words as keys the commonly occurring suffixes such as ing, tion, ain, ed and ly seldom span more than one item. Such chains waste both storage space and search time. Indeed, if a node governs only a few items, instead of continuing the tree in the normal manner from that node-- entailing the use of several levels to reach the items-- the items should be assigned to the nodes of the filial set of that node and no additional levels used. By doing this, fewer nodes are traversed to reach the leaves, thereby reducing both the required storage space and the search time. A variable length value field is required for these leaf nodes, however, because all the key elements of the item not already assigned in the partial key of the parent node must be assigned the values of the leaves.

Thus, in this case it is of interest to determine at which nodes the branching should be terminated to achieve a more efficient search time and a reduced storage requirement. A path leading to a block of items is considered to have its optimum length if the expected search time for items in the block is increased when the path length is changed. The computation of this optimum path length is based on the number of items, g, which a node x governs and the number of nodes, m, in its filial set. The nodes of the filial set of x each govern an average of g/m items. If the branching is discontinued at node x, the expected search time x is

$$t_d = \tfrac{1}{2}(g + 1).$$

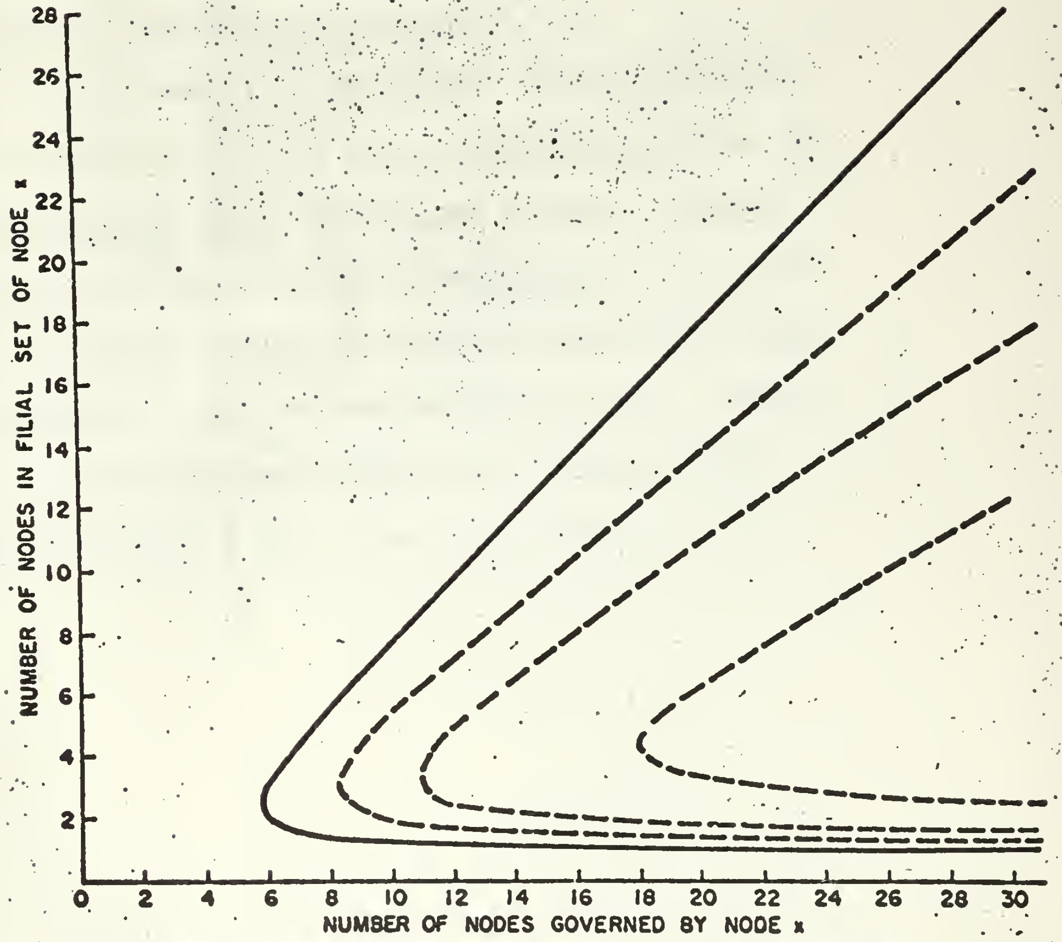
If, however, the branching is continued for one more level, the expected search time from x is

$$t_c = \tfrac{1}{2}\left(m + \frac{g}{m}\right) + 1.$$

Thus, the expected search time from x will decrease if branching is continued when $t_d > t_c$. [37]

Figure 15 displays the relation between t_d and t_c ; for any node with its (g,m) lying in the pie-shaped region to the right of the solid curve, the search time can be decreased by continuing the branching from that node. The dashed curves indicate the relative improvement in the search time between the cases of continuing and discontinuing the branching from the node. As $g \geq m$, the criterion as to whether to branch or not given in Figure 15 may be approximated by stating that the branching should be continued from any node which governs more than six items. [37]

Given a situation where the size of h and m are either (or both) required to be larger than optimum, we are then faced with a tree which is to some degree vacant. An analysis of the manipulations and searches associated with such a tree forces one to rely primarily on the mathematics of probability. Scidmore and Weinberg have developed an analysis of trees from this viewpoint. [33] The analysis includes the situations 1) where keywords are chosen randomly from those with lengths of one through nine elements, and

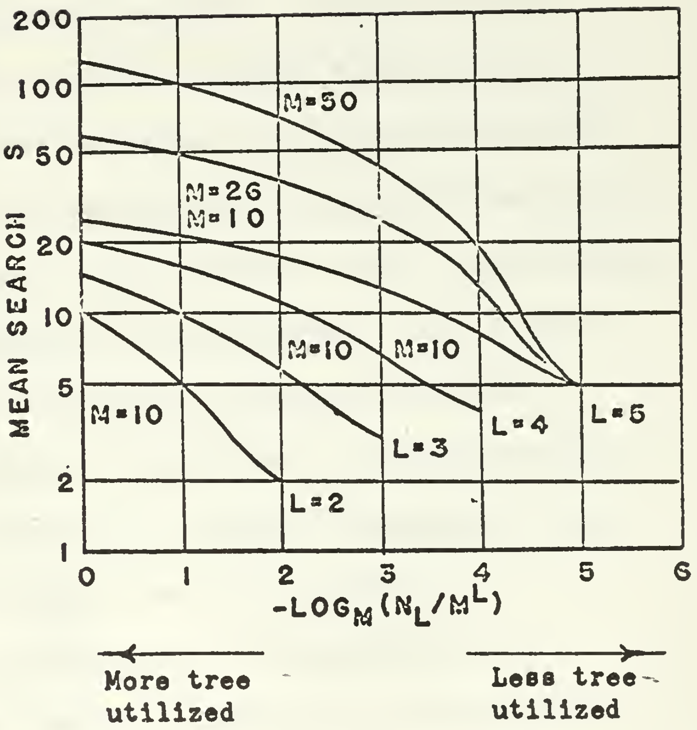


Relative Search Time as a Function of Filial Set Size and Number of Governed Nodes

Figure 15

2) where all keywords are the same length. Values of $m = 10$, 26, and 50 were used to simulate decimal numbers, alphabetic characters, and alphanumeric characters respectively.

In the case where the keywords were considered all the same length, the results of the analysis for the predicted mean search S correspond exactly with results previously derived for t (recall that in our previous development we considered t as a direct function of the number of item times, i. e. the number of links traversed. t is therefore identical to S). A plot of <u>the mean search</u> vs. <u>the logarithm of the fraction of the tree in use</u> is shown in Figure 16 for several values of m and L (L here is equivalent to h). It is apparent that when there are very few sequences stored, the mean search is approximately L and somewhat independent of m. However, as the number of stored sequences approaches m^L (same as m^h), the mean search approaches $\frac{1}{2} L(m + 1)$ as noted before.



Mean search for the fixed sequence length (L) and filial set size (M) as shown.

Figure 16

8. Comparison of modified trees.

A useful modification to the more straightforward tree structures already described is the Trie.[1] Fredkin has proposed a scheme in which the heirs are chained and the sib sets are arranged in blocks. [8] The basic feature is that whenever a node on level i requires an heir on level i + 1, a block large enough to contain an entire filial set is established. The heir address in level i is the address of the first logical member of the filial set and all other members of that set are understood to exist in sequence in the succeeding memory locations. Therefore, each computer word corresponding to a node need only contain the heir address since the node's label is implicit in its position within the sib set. The advantage in machines and applications where word size is a problem is obvious. Also, retrieval from a trie structure requires one indexing manipulation for each level of the tree, not an item-by-item search of the nodes as in the tree structure. Thus, the expected search time is proportional to the average path length, h, rather than $\frac{1}{2}h\ (m + 1)$. This speed advantage is compensated by requiring more storage locations, however.

The trie structure is introduced at this point since it represents a middle-ground compromise between the two extremes in tree structures, i.e. the balanced tree and the completely elided unbalanced tree. It therefore provides a fulcrum for comparison

[1]The word trie is taken from the word retrieval.

[illegible]

of the various types. See Figure 17 for a pictorial comparison between these three basic types under a storage condition where the majority of the balanced tree is vacant.

Let d be the average number of elements utilized in each filial set where d is a subset of the m possible elements in a filial set. Note that in a balanced tree, d = m. Computing the number of storage locations W, we find:

For a balanced tree

$$W = \sum_{i=1}^{h} m^{i} = m \frac{m^{h} - 1}{m - 1}$$

For a trie

$$W = m + dm + d^{2}m + \dots + d^{h-1} m = m \frac{d^{h} - 1}{d - 1}$$

For an elided unbalanced tree

$$W = \sum_{i=1}^{h} d^{i} = d \frac{d^{h} - 1}{d - 1}$$

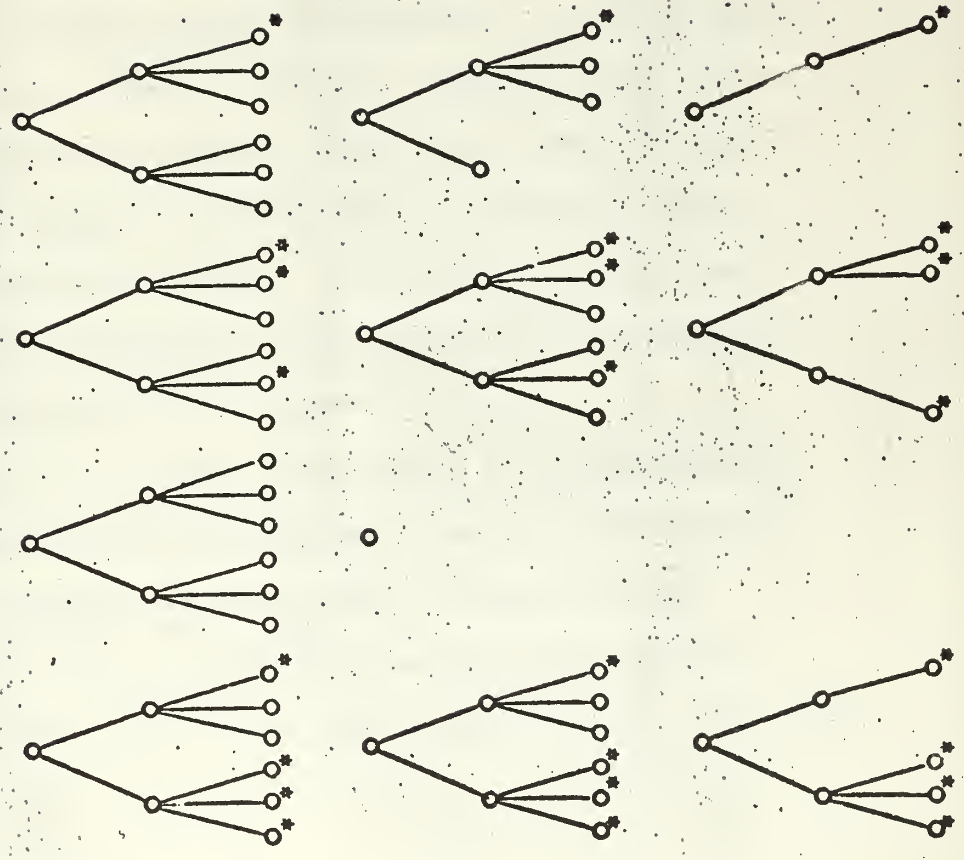
Since d < m,

$$d \frac{d^{h} - 1}{d - 1} < m \frac{d^{h} - 1}{d - 1} < m \frac{m^{h} - 1}{m - 1}$$

or

$$W_{\text{elided}} < W_{\text{trie}} < W_{\text{balanced}}$$

From this simple comparison, it is obvious that the ratio of d/m is a convenient index to the preferred tree structure. This index, however, is sensitive only to the storage criteria and neglects the equally important factor of search time. One of the



Partially vacant balanced tree

Trie

Elided tree

Comparison of the extremes in tree structure and a compromise (mean) type tree.

Figure 17

advantages of the trie is a faster search time so that it is even more attractive than the elided tree in certain instances.

The balanced tree finds its place in applications such as the multi-list structure developed at the University of Pennsylvania. [18] The tree acts as a translator whose inputs are coded information words and whose outputs are addresses of lists on which items of information are stored. The keys, assumed to enter the system in random order, are inserted in the tree so that the monotonic order of key values is preserved and the balanced growth of the tree is continuously maintained. The leaves of the tree lead into the Multi-Association Area as lists. Each list is a string of items that have at least on key in common.

A technique which is roughly equivalent to an h-level balanced tree is the addressing of multidimensional arrays as linear arrays or vectors. This technique provides a means for partitioning a file into subfiles which is simpler than the tree structure and also permits more rapid entry into a subfile than the tree. [14] [12]

9. Fibonaccian searching.

The standard binary search of an ordered list for a given argument consists of comparing the argument with the middle item of the list in order to isolate the item desired to a list half as long as the original list. This process is repeated until the item is found. As we have seen, the number of trials required is of order $\log_2 n$ for a table of n entries. If it is desired to save memory space, the addresses for the successive trials are computed, but this requires time consuming division or multiplication by an inverse in machines having no binary shift. A table of the powers of two is stored and used if a program which is conservative of time is desired.

One method to conserve both time and storage space utilizes the Fibonacci numbers defined by:

$$\begin{aligned} u_i &= i, & i &< 2 \\ u_i &= u_{i-1} + u_{i-2}, & i &\geq 2 \end{aligned}$$

The important distinction here is that the successive increments are found by subtraction. [6]

If at some point in the process the item has been isolated to an interval of size u_i beginning at A, compare x to C_{i-1} where

x = value of the argument

C_{i-1} = contents of $A + u_{i-1}$

If $x < C_{i-1}$, the item is now isolated to an interval of size u_{i-1} beginning a A, hence replace i by i-1 and repeat the process.

If $x > C_{i-1}$, the item is now isolated to an interval of size $u_i - u_{i-1} = u_{i-2}$ beginning at $A + u_{i-1}$, so replace i by i - 2, A by $A + u_{i-1}$ and repeat the process.

It is easily verified by induction that

$$u_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

where $$\phi = \frac{1+\sqrt{5}}{2}$$

For large n,

$$u_n \rightarrow \frac{\phi^n}{\sqrt{5}}, \qquad \frac{u_n}{u_{n-1}} \rightarrow \phi = 1.618 \ldots,$$

and $$\frac{u_n}{u_{n-2}} \rightarrow \phi^2 = 2.618 \ldots$$

In the worst case, the reduction factor R_1 (defined as the ratio of successive intervals) is ϕ, requiring

$$\log_\phi m = \frac{\log_2 m}{\log_2 \phi} \text{ trials.}$$

In the first case (if x is not found), R_2 is ϕ^2, requiring

$$\log_{\phi^2} m = \frac{\log_2 m}{2 \log_2 \phi} \text{ trials}$$

Clearly, the expected value of R is $pR_1 + qR_2$, where p is the probability of R_1 and q is the probability of R_2.

$$p = \frac{1}{\phi} \quad \text{and} \quad q = \frac{1}{\phi^2}$$

giving

$$R = \left(\frac{1}{\phi}\right)\phi + \frac{1}{\phi^2}(\phi^2) = 2$$

as in the binary search.

Therefore, for searching an ordered list having n elements, the expected searching time is of order $\log_2 n$ and the maximum searching time is of order $\log_\phi n$. [6]

BIBLIOGRAPHY

1. Bush, Vannevar, As we may think. The Atlantic Monthly, July 1945.

2. Clampett, H.A. Jr. Randomized binary searching with tree structures. Communications of the Association for Computing Machinery, March, 1964: 163.

3. Collins, G. E. Method of overlapping and erasure of lists. Communications of the Association for Computing Machinery, Dec., 1960: 655.

4. Dineen, G. P. Programming pattern recognition. Proceedings of the 1955 Western Joint Computer Conference, Institute of Radio Engineers, March, 1955.

5. Feldman, J. TALL--List processor in the Philco 2000. Communications of the Association for Computing Machinery, Sept., 1962: 484.

6. Ferguson, D. E. Fibonaccian searching. Communications of the Association for Computing Machinery, Dec., 1960: 648.

7. Flynn, M.J. and Henderson, D.S. Variable field-length manipulation in a fixed word-length memory. IEEE Transactions on Electronic Computers, Oct., 1963: 512.

8. Fredkin, E. Trie memory. Communications of the Association for Computing Machinery, Sept., 1960: 490.

9. Galernter, H. Realization of a geometry theorem proving machine. Proceedings of the International Conference on Information Processing, 1959.

10. Galernter, H., Hansen, J.R., and Gerberich, C.L. A FORTRAN-compiler list-processing language. Journal of the Association for Computing Machinery, April, 1960: 87.

11. Galernter, H., and Hansen, J.R. Intelligent behavior in problem-solving machines. IBM Journal of Research and Development, October, 1958.

12. Gower, J.C. The handling of multiway tables on computers. Computer Journal, Jan., 1962: 280.

13. Harvard University, Summary of contract research. The Computation Laboratory, Harvard University, Scientific Report No. ISR-4 (Final Report), Aug., 1963.

14. Hellerman, H. Addressing multidimensional arrays. Communications of the Association for Computing Machinery, April, 1960: 205.

15. Hibbard, T.N. Some combinatorial properties of certain trees with applications to searching and sorting. Journal of the Association for Computing Machinery, Jan., 1962: 13.

16. Iverson, K. E. A programming language. John Wiley and Sons, Inc., New York and London, 1962.

17. Landauer, W.I. The balanced tree and its utilization in information retrieval. IEEE Transactions on Electronic Computers, Dec., 1963: 863.

18. Landauer, W.I. Construction and expansion of a coding tree. The Multi-List System, Technical Report No. 1, Part 1, Volume I, Chapter 2, University of Pennsylvania, The Moore School of Electrical Engineering, Nov., 1961.

19. Marma, V. The machine representation of abstract trees in information retrieval. Scientific Report No. ISR-1, Section II, Information Storage and Retrieval, The Computation Laboratory, Harvard University, Cambridge, Mass., Nov., 1961.

20. Maron, M.E. Probability and the library problem. Behavioral Science, July, 1963: 250.

21. Maron, M.E. and Kuhns, J.L. On relevance, probabilistic indexing and information retrieval. Journal of the Association for Computing Machinery, July, 1960: 216.

22. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Communications of the Association for Computing Machinery, April, 1960: 184.

23. Muth, V.O. and Scidmore, A.K. A memory organization for an elementary list processing computer. IEEE Transactions on Electronic Computers, June, 1963: 262.

24. Newell, A. Information processing language - V manual. Prentice-Hall, 1961.

25. Newell, A. and Simon, H.A. The logic theory machine. IRE Transactions on Information Theory, Sept., 1956.

26. Newell, A., Shaw, J.C. and Simon, H.A. Empirical exploration of the logic theory machine. Proceedings of the 1959 Western Joint Computer Conference, Institute of Radio Engineers, Feb., 1957.

27. Newell, A. and Tonge , F.M. An introduction to information processing language V. Communications of the Association for Computing Machinery, April, 1960: 205.

28. Newhouse, V.L. and Fruin, R.E. A cryogenic data addressed memory. Proceeding of the 1962 Spring Joint Computer Conference, American Federation of Information Processing Societies, Vol. 21, May, 1962.

29. Perlis, A.J. and Thornton, C. Symbol manipulation by threaded lists. Communications of the Association for Computing Machinery, April, 1960: 195.

30. RCA Review. Superconductive associative memories. Volume 24, Sept., 1963: 325.

31. Ross, D.T. A generalized technique for symbol manipulation and numerical calculation. Communications of the Association for Computing Machinery, March, 1961: 147.

32. Salton, G. Manipulation of trees in information retrieval. Communications of the Association for Computing Machinery, Feb., 1962: 103.

33. Scidmore, A.K. and Weinberg, B.L. Storage and search properties of a tree-organized memory system. Communications of the Association for Computing Machinery, Jan., 1963: 28.

34. Selfridge, O.G. Pattern recognition and modern computers, Proceedings of the 1955 Western Joint Computer Conference, Institute of Radio Engineers, Feb., 1957.

35. Sherman, P.M. Programming and coding digital computers. John Wiley and Sons, Inc., New York and London.

36. Sussenguth, E.H.Jr., An evaluation of storage allocation systems. Scientific Report No. ISR-2, Section V, Information Storage and Retrieval, The Computation Laboratory, Harvard University, Cambridge, Mass., Sept., 1962.

37. Sussenguth, E.H.Jr. Use of tree structures for processing files. Communications of the Association for Computing Machinery, May, 1963: 272.

38. Swets, J.A. Information retrieval systems: Statistical decision theory may provide a measure of effectiveness better than measures proposed to date. Science, July,1963: 245.

39. Weizenbaum, J. Knotted list structures. Communications of the Association for Computing Machinery, March 1962: 161.

40. Weizenbaum, J. Symetric list processor. Computer Laboratory, General Electric Company, Sunnyvale, Calif., Feb., 1963.

41. Wigington, R.L. A machine organization for a general purpose list processor. IEEE Transactions on Electric Computers, Dec., 1963: 707.

APPENDIX

CLOSED FORMS OF SUMMATIONS

$$(1) \qquad \sum_{k=1}^{a} c^k = c\frac{c^a - 1}{c - 1}$$

$$(2) \qquad \sum_{k=1}^{a} \frac{k}{c^k} = \frac{c(c^a - a - 1) + a}{c^a(c - 1)^2}$$

$$(3) \qquad \sum_{k=1}^{a} k\, c^k = \frac{c^{a+1}(ca - a - 1) + c}{(c - 1)^2}$$

All of these expressions may be verified by induction. For example, in (1) let

$$S_i = \sum_{k=1}^{i} c^k.$$

Then $S_1 = c$ and $S_i = S_{i-1} - c^i$ must be satisfied.

(2) may be derived from (1) by the following manipulation:

$$(1 - \frac{1}{b}) \sum_{1}^{a} \frac{k}{b^k} = \sum_{1}^{a} \frac{k}{b^k} - \sum_{1}^{a} \frac{k}{b^{k+1}}$$

$$= \sum_{1}^{a+1} \frac{k}{b^k} - \sum_{1}^{a+1} \frac{k-1}{b^k} - \frac{a+1}{b^{a+1}}$$

$$= \sum_{1}^{a+1} \frac{1}{b^k} - \frac{a+1}{b^{a+1}}$$

(3) may be derived from (2) directly.